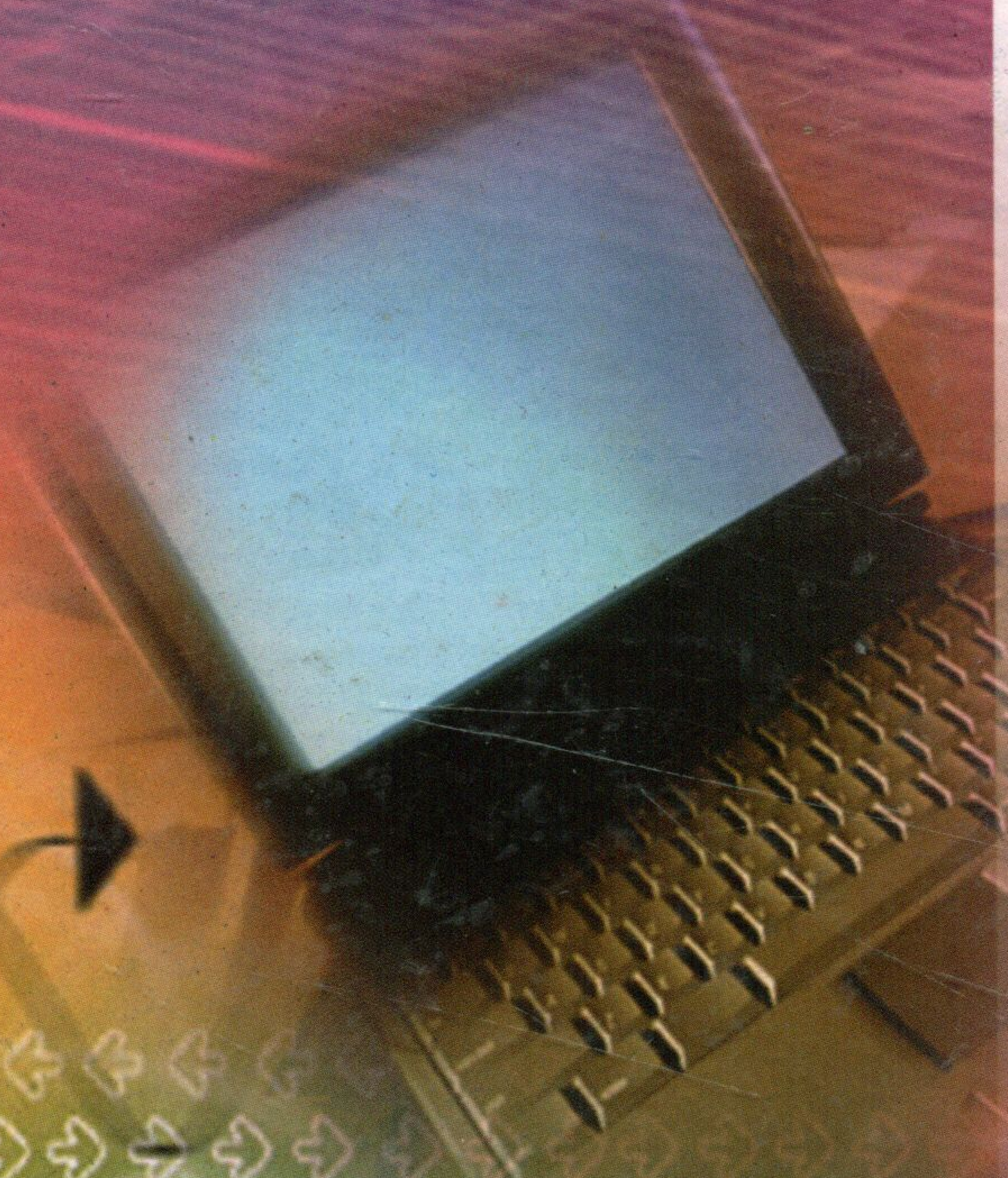


تركيب البيانات

وتصميم الخوارزميات



د. سليمان مصطفى
د. فواز الزغلول
د. خليل الهندي
د. نجيب الكوفحي

الشركة العربية المتحدة للتسويق والتوريدات

تركيب البيانات
وتصميم الخوارزميات

إعداد

د. سليمان مصطفي   د. فواز الزغول

د. خليل الهندي   د. نجيب الكوفحي

2010

الكـــــتاب : تركيب البيانات وتصميم الخوارزميات

المـــــؤلف :سليمان مصطفي وأخرون

الطبعة الأولـــي : القاهرة 2010

رقم الإيــــداع : 9177/ 2009

I .S .B .N : 978-977-477-023-4

العنـــــوان :ص.ب: 203 مكتب بريد هليوبوليس - مصر الجديدة- 11757

القاهرة - جمهورية مصر العربية

البريد الإلكترونـي u_ara@yahoo.com:

الموقع الإلكتروني www.uarab.net:

موبايـــــل : 022-010-1763677 /002-010-3401184

مصطفي ، سليمان

تركيب البيانات وتصميم الخوارزميات/ مصطفي، واخرون

ط 1. - القاهرة: الشركة العربية المتحدة للتسويق والتوريدات، 2010

520ص؛17×24سم

تدمـــك 4- 23-477-977-978

1- الخوارزامية - برمجة حاسبات

أ- مصطفي، سليمان (مؤلف مشارك)

المقدمة

عزيزي الدارس، أهلاً بك إلى كتاب «تركيب البيانات وتصميم الخوارزميات». إن موضوع تراكيب البيانات على درجة بالغة الأهمية في علم الحاسوب، حيث أنه يناقش الطرق المختلفة لتنظيم البيانات داخل الحاسوب. ومن المهم جداً، عزيزي الدارس، اختيار الطريقة المناسبة لتنظيم البيانات لعدة أسباب منها:

1. أن اختيار الطريقة المناسبة لتمثيل البيانات يؤدي إلى استخدام خوارزميات حل أكثر كفاءة (من حيث وقت التنفيذ وحجم ذاكرة الحاسوب المستخدمة. ومن هنا يكون الترابط بين موضوع تراكيب البيانات وتحليل الخوارزميات، (لمعرفة وقت التنفيذ وحجم ذاكرة الحاسوب اللازمين) وكثيراً ما يقال إن:

البرنامج = تركيبة بيانات + خوارزمية الحل.

2. أن اختيار الطريقة المناسبة لتمثيل البيانات يجعل البرنامج أيسر للكتابة، حيث تكون خوارزمية الحل أوضح وأقرب إلى الواقع مما يؤدي إلى تقليل الوقت والجهد اللازمين لكتابة البرنامج.
3. أن وضوح ومنطقية طريقة الحل تؤدي إلى برامج واضحة يسهل فهمها وتعديلها عند الحاجة في وقت قصير.

مما سبق يتضح لك أن اختيار تركيبة البيانات المناسبة خطوة مهمة جداً لتحقيق أعلى مستويات الكفاءة، ليس فقط كفاءة الحاسوب، وإنما أيضاً كفاءة المبرمج في كتابة وتعديل البرنامج عند الحاجة.

نتمنى لك، عزيزي الدارس، كل التوفيق في قراءة هذا الكتاب كما نتمنى منك إعطاءه الوقت والجهد الذي يستحقه.

ويتألف هذا الكتاب من إحدى عشرة وحدة على النحو الآتي:

الوحدة الأولى: (مقدمة إلى تراكيب البيانات): وتهدف هذه الوحدة إلى توضيح المفاهيم الأساسية المتعلقة بتراكيب البيانات.

الوحدة الثانية: (مبادئ تحليل الخوارزميات): وتهدف هذه الوحدة إلى تعريفك أسس كتابة الخوارزميات وتحليلها وتقييمها، كما تمكنك هذه الوحدة من بناء الخوارزميات المختلفة وتصميمها.

الوحدة الثالثة: (التراكيب التجريدية): وتهدف هذه الوحدة إلى إكسابك مهارة استخدام تراكيب البيانات بطريقة تجريدية.

الوحدة الرابعة: (القوائم): وتهدف هذه الوحدة إلى توضيح موضوع القوائم وهو أحد أهم تراكيب البيانات.

الوحدة الخامسة: (السلاسل الرمزية): وتهدف هذه الوحدة إلى إبراز تراكيب البيانات المختلفة التي يمكن استخدامها لتمثيل السلاسل الرمزية واستخدامها بطريقة تجريدية.

الوحدة السادسة: (الطوابير): تناقش هذه الوحدة مفهوم الطوابير التي تعتبر من تراكيب البيانات المهمة، وأسلوب تمثيلها في ذاكرة الحاسوب، وإجراء عملية الإضافة والحذف منها، والاستخدامات والتطبيقات المختلفة لها.

الوحدة السابعة: (المكدسات): وتهدف هذه الوحدة إلى تعريفك بنوع آخر من تراكيب البيانات ألا وهو المكدسات وهي نوع خاص من القوائم حيث تتم الإضافة والحذف فيها من نفس الجهة.

الوحدة الثامنة: (الاستدعاء الذاتي): إن العديد من تراكيب البيانات تتطلب استدعاء البرنامج الفرعي لنفسه عدة مرات لمعالجته، وعليه فإن هذه الوحدة ستعرفك الجوانب المختلفة المتعلقة بالاستدعاء الذاتي.

الوحدة التاسعة: (الهياكل الشجرية): تناقش هذه الوحدة الأنواع المختلفة للهياكل الشجرية وهي تراكيب بيانات غير خطية. وتستخدم لأغراض مختلفة منها تمثيل البيانات ذات العلاقة الهرمية في البحث والفهرسة والفرز وفي التطبيقات المختلفة الأخرى.

الوحدة العاشرة: (المخططات والشبكات): وتهدف هذه الوحدة تعريفك بنوع آخر من تراكيب البيانات غير الخطية وهي المخططات والشبكات واللتان تستخدمان في تمثيل العديد من البيانات المستخدمة في الحياة اليومية.

الوحدة الحادية عشرة: (الفرز والبحث): تعالج هذه الوحدة موضوع الفرز والذي يعني ترتيب مجموعة من العناصر حسب قيمة مفتاح معين، وذلك إما تصاعدياً أو تنازلياً. كما تبحث موضوع البحث بطرقه الثلاث الرئيسة وهي البحث التتابعي والبحث الثنائي والبحث المفهرس.

والله ولي التوفيق

محتويات الكتاب

الوحدة
الأولى

مقدمة إلى تراكيب البيانات

محتويات الوحدة

1. المقدمة

1.1 تمهيد

عزيزي الدارس، أهلاً بك إلى الوحدة الأولى من كتاب «تركيب البيانات وتصميم الخوارزميات» وهي بعنوان «مقدمة إلى تراكيب البيانات».

تهدف هذه الوحدة إلى إعطائك لمحة سريعة عن الأنواع المختلفة للبيانات، وتوضيح بعض العمليات الأساسية وبيان أهمية اختيار تركيب البيانات المناسبة.

تتناول هذه الوحدة تعريف مفهوم البيانات وإعطاء لمحة مختصرة عن الأنواع المختلفة للبيانات، وطرق تمثيلها في الذاكرة. ثم توضح الوحدة كيفية التفريق بين مفهوم التركيب المنطقي، ومفهوم التركيب الفيزيائي للبيانات. ويتلو ذلك، مناقشة أهمية التراكيب البيانية وأهم العمليات الأساسية التي تجري عليها، والأسباب التي تبرر اختيار تركيب معين دون غيره أحياناً.

وقد تم تزويد هذه الوحدة بالعديد من الأمثلة التوضيحية، والتدريبات، وأسئلة التقويم الذاتي، جنباً إلى جنب مع بعض الرسومات والأشكال التي توضح بعض المفاهيم. وستجد، عزيزي الدارس، حلولاً للتدريبات التي تصادفها في نهاية الوحدة.

فأهلاً وسهلاً.

2.1 أهداف الوحدة

ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1. تبين أنواع البيانات المختلفة وكيفية تخزينها.
2. تذكر بعض أنواع تراكيب البيانات.
3. تشرح أهمية تراكيب البيانات المختلفة.
4. تذكر بعض العمليات المهمة المستخدمة على تراكيب البيانات.

3.1 أقسام الوحدة

تتكون هذه الوحدة من ثلاثة أقسام رئيسة. ترتبط بقائمة الأهداف السابقة. فالقسم الأول يبحث في البيانات وأنواعها، وكيفية تخزينها. وبذلك يتحقق الهدف الأول من أهداف الوحدة. أما القسم الثاني فيشرح التركيب المنطقي والفيزيائي للبيانات، ويرتبط بدوره بالهدف الثاني. بعد ذلك نأتي إلى توضيح البيانات المختلفة وأهميتها والعمليات المنفذة عليها، وهذا يشكل القسم الثالث ويحقق الأهداف الثاني والثالث والرابع من أهداف الوحدة.



4.1 القراءات المساعدة

لقد قدمنا كثيراً من المفاهيم الأساسية المذكورة في هذه الوحدة بشكل مبسط ومختصر. ولهذا، فمن المفيد، عزيزي الدارس، أن تعود إلى المصادر التالية لما تتضمنه من مادة علمية إضافية، وأمثلة وإيضاحات ستساعدك على استيعاب المادة التي تضمنتها هذه الوحدة. وهذه المصادر هي:

1. Beidler, John, An Introduction to Data Structures. Boston: Allyn And Bacon, 1982, pp. 1-13, & pp. 15-36.
2. Lipschutz, Seymour, Schaum's Outline of Theory and Problems of Data Structures. New York: McGraw-Hill, 1986, pp. 1-16 & pp. 17-40.
3. Tremblay, J.P, and Sorenson, P.B., An Introduction to Data Structures with Applications. New York: McGraw-Hill, 1976. pp. 9-59.

2. البيانات وأنواعها (Data And Data Types)

نصادف في حياتنا اليومية، عزيزي الدارس، العديد من المواقف والأشياء التي نحتاج إلى وصفها بطريقة معينة. وبطبيعة الحال فإن هذا الوصف يختلف باختلاف الموقف أو الشيء الذي نرغب في وصفه. ومن هنا فإننا نستخدم حروفاً أو كلمات لتسمية الأشياء والأفعال المختلفة أو وصفها. ونستخدم الأرقام للقيام بالقياس والعد، ونجيب بنعم أو لا أو ما يفيد الصواب والخطأ على بعض المواقف والاستفسارات. فهذه الحروف والكلمات والأرقام والإجابات هي ما نشير إليه بمصطلح البيانات. فالبيانات بهذا المعنى، هي مجموعة من القيم، وظيفتها التعبير عن الكينونات التي نتعامل معها والأحداث التي نعيشها. وتختلف البيانات في خصائصها وأنماطها باختلاف الكينونات أو الأحداث التي تعبر عنها، وبالتالي فإنها متباينة فيما بينها من حيث نوع العمليات التي تجري عليها، حسابية أو منطقية أو غير ذلك.

وبناءً على هذا المفهوم نقول مثلاً إن لدينا بيانات عددية عندما نتعامل مع الأرقام، وبيانات رمزية عندما نتعامل مع الحروف والكلمات، وبيانات منطقية عندما نتعامل مع بيانات تحتمل إجابات الصواب والخطأ. فهذه جميعاً يمكن أن نصطلح على تسميتها باسم أنماط بيانية بسيطة (Simple data types). وهناك في المقابل العديد من الحقائق المختلفة التي يصعب متابعتها والإحاطة بجميع جزئياتها أحياناً. ومرد هذه الصعوبة تعدد البيانات الذي ينطوي عليه وصف هذه الحقائق. ولكي نقوم بتذليل هذه الصعوبة نقوم بتنظيم هذه البيانات في مجموعات، وجداول، وقوائم، وسجلات، وملفات وغير ذلك من أساليب التنظيم والتركيب المختلفة. ونقوم بتخزين هذه البيانات بطريقة يمكننا معها تذكر مواضعها ومن ثم استرجاعها حين الحاجة. وبهذه الطريقة نخفف عن أنفسنا عبء تذكر كل عنصر من عناصر البيانات على حدة. والمقصود بعناصر البيانات هنا تلك الحقائق الفردية التي يتم الجمع بينها وفق أحد أساليب التركيب والتنظيم. وهذا النوع من البيانات هو ما نشير إليه بالاسم أنماط بيانية مركبة (Structured data types).

وبينما تحرص لغات الحاسوب على تضمين جميع الأنماط البيانية البسيطة أو معظمها، فإنها لا توفر إلا بعض الأنماط البيانية المركبة كالسجلات والمصفوفات والملفات. وعلى سبيل المثال فإن الجدول (1) يبين الأنماط البيانية التي تشتمل عليها لغة C++ التي سبق أن درستها في مقرر منفصل. ولكن هذه الأنماط البيانية المركبة التي توفرها اللغة، وإن كانت مهمة في كل التطبيقات العملية، لا يمكن التعبير بها عن كل المشكلات التي تنطوي

على بيانات متعددة. ومن هنا، فإن مثل هذه المشكلات تحتاج نوعاً خاصاً من المعالجة وجهداً خاصاً من المبرمج.

جدول (1): الأنماط البيانية المتوفرة في لغة ++C

أنماط بيانية بسيطة (Fundamental)				أنماط بيانية مركبة (Derived)			
void				constant			
Arithmetic				array			
Floating	Integral			function			
float	unsigned	signed	enum	Indirect:	reference	pointer	
	char	char		Structure:	class	struct	union
double	short	short					
long double	int	int					
	long	long					

وعلى هذا الأساس فإن المصطلح مركب أو تركيب بياني (data structure) يشير بشكل مباشر إلى مجموعة من عناصر البيانات التي لها تنظيم خاص. ومن ملامح هذا التنظيم وجود أسلوب خاص للوصول إلى العناصر الفردية، سواءً خلال عملية تخزين القيم أو خلال عملية الاسترجاع. وبناءً على هذا المفهوم فإن هناك ميزتين أساسيتين تتسم بهما **التراكيب البيانية وهما:**

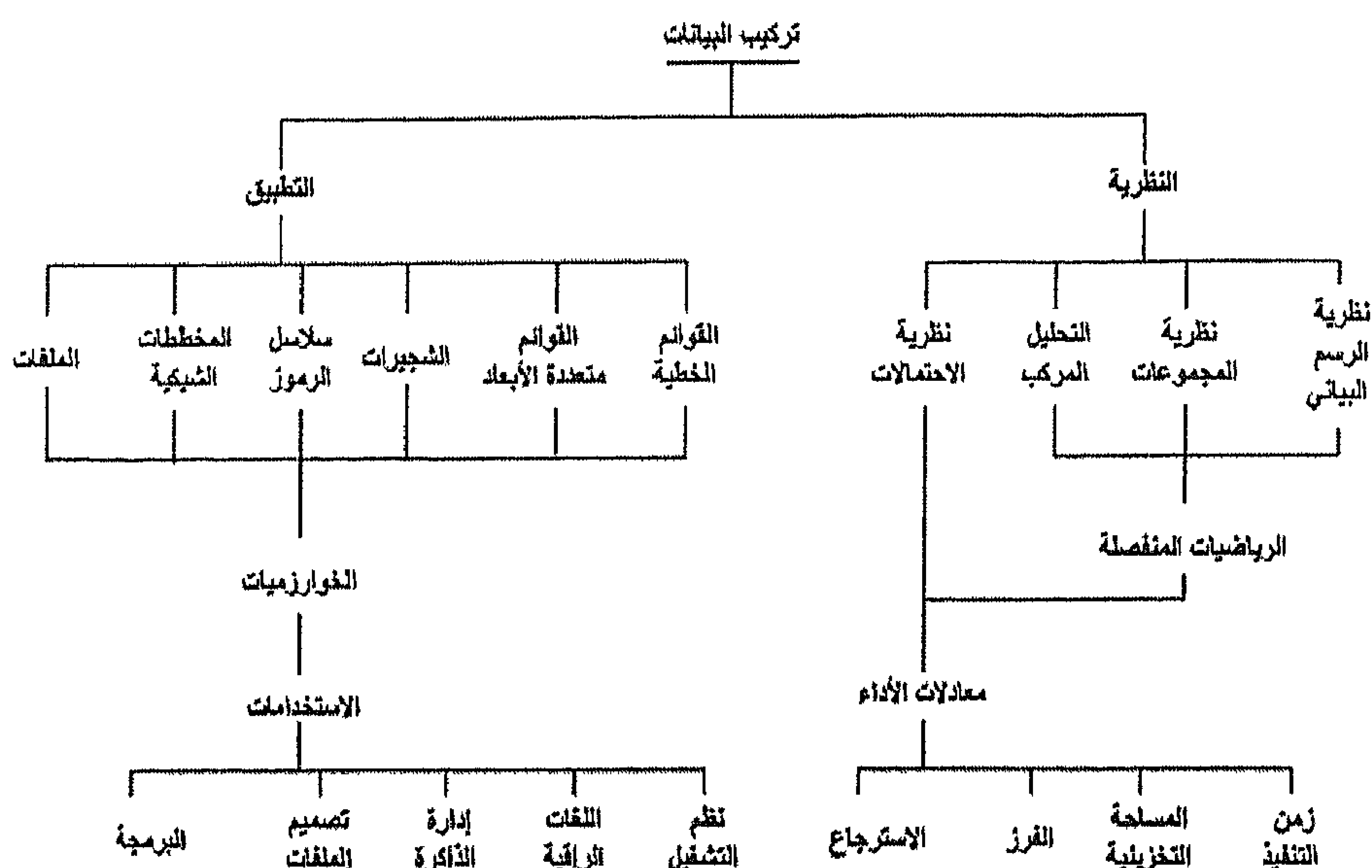
أ. يمكن التعامل مع كل عنصر منها على حدة، كما لو كان متغيراً مستقلاً كبقية المتغيرات.

ب. طريقة تنظيم العناصر تؤثر مباشرة على طريقة الوصول إلى العنصر الواحد أو التعامل مع المركب البياني ككل.

أما المصطلح تركيب البيانات (data structures) من جهة أخرى، فإنه يشير إلى الطرق والأساليب المختلفة التي يمكن من خلالها ترجمة التصور المنطقي للبيانات (كما يراها المبرمج) إلى تمثيل مناسب في ذاكرة الحاسوب. فهو، بهذا المعنى، يهتم بالطرق المختلفة لتنظيم البيانات، وبالخوارزميات اللازمة لمعالجة هذه البيانات في ذاكرة الحاسوب. وكما يتضح من الشكل (1)، فإن لهذا الموضوع جانبين: أحدهما نظري والآخر تطبيقي. فالجانب النظري يتناول الأسس المنطقية والرياضية التي تقوم عليها عملية تركيب البيانات وتحليل كفاءة التراكيب البيانية من حيث المساحة التي تحتلها في الذاكرة

والزمن الذي يستغرقه تنفيذ الخوارزميات الخاصة بها. أما الجانب التطبيقي فيتناول الأسس العملية والبرمجية لإيجاد العلاقة التنظيمية بين العناصر المختلفة للتراكيب البيانية وإيجاد السبل المناسبة للوصول إلى هذه العناصر.

وكما يمكن أن تتوقع فإن العناصر الفردية للتراكيب البيانية لا بد وأن تتكون في النهاية من أنماط بيانية بسيطة من النوع الذي أشرنا إليه سابقاً.



الشكل (1): الجوانب المختلفة لدراسة تركيب البيانات
(مقتبس من: Lewis & Smith)

مهما اختلفت طرق التركيب أو تعددت مستوياته فقد نخزن في هذه العناصر قيماً:

- عددية: صحيحة (integer) أو حقيقية (real).
- رمزية، كالحروف وغيرها من الرموز.
- منطقية (logical).
- تعدادية أو جماعية (enumerated).
- إشارية (pointer type)، كعناوين مطلقة لمواضع الذاكرة (absolute) أو عناوين نسبية (relative).

قد تكون هذه القيم مزيجاً من القيم التي تنتمي إلى أنماط بيانية مختلفة وفق ما تمليه طبيعة المركب البياني الذي نحن بصدد التعامل معه في البرنامج. وقبل أن نبدأ بمناقشة التراكيب البيانية، سنحاول، عزيزي الدارس، فيما يلي التعريف بهذه الأنواع البيانية البسيطة وكيفية تخزينها في ذاكرة الحاسوب، فهي تمثل أدنى مستويات التركيب في الذاكرة، وهناك من يشير إليها أحياناً باسم التراكيب البيانية البدائية (primitive).

1.2 الأعداد الصحيحة (Integers)

إن الأعداد الصحيحة والعمليات التي تجري عليها (كالجمع والطرح والضرب والقسمة وغيرها) مألوفة لنا جميعاً. وهي معرفة ضمن المجموعة التالية:

{...,n+1), -n,..., -2, -1, 0, 1, 2,..., n, n+1) , ...}

والغاية من استخدامها تتمثل في عملية العد الاحصائي للحالات أو الأشياء التي نتعامل معها.

ويتم تخزين الأعداد الصحيحة في ذاكرة الحاسوب باستخدام إحدى طريقتين: الأولى هي الطريقة الثنائية (binary)، والثانية هي الطريقة العشرية (decimal). وفيما يلي توضيح لما تعني كل منهما:

أ - التمثيل الثنائي (binary representation):

تخزن الأعداد الصحيحة في ذاكرة الحاسوب على شكل ثنائي وذلك لتسهيل العمليات الحسابية على هذه الأرقام. وبصفة عامة فإن العدد الصحيح يخزن في موضع واحد (word) من مواضع الذاكرة. إلا أن هناك بعض لغات الحاسوب (مثل PL/I والأسمبلي) التي تتيح للمبرمج إمكانية تخزين العدد الصحيح في جزء من الموضع. والحد الأدنى في هذه الحالة هو بايت واحد. وفي هذه الحالة يضطر الجهاز إلى نقل القيمة إلى مخزن مؤقت من أجل إعادته إلى وضعه الأصلي لتجري عليه العملية الحسابية والمنطقية المطلوبة. ومن أساليب التخزين المتبعة لهذه الغاية طريقة المكمل الثاني (2s complement).
ويحسب المكمل الثاني كما يلي:

1. يحول الرقم إلى النظام الثنائي.
2. يحسب المكمل الأول عن طريق تحويل كل 1 إلى صفر وكل صفر إلى 1
3. نضيف 1 إلى المكمل الأول لنحصل على المكمل الثاني.

فمثلاً نخزن الرقم (9613) مرة بصيغة الموجب ومرة بصيغة السالب باستخدام المكمل الثنائي (2's complement) كما يلي: متم

(9613+)   0010010110001101

(9613-)   1101101001110011

مع العلم بأن الخانة الثنائية الأخيرة على اليسار تدل على إشارة الرقم: فالصفر يدل على أن الرقم موجب والواحد يدل على أن الرقم سالب.

ب - التمثيل العشري (decimal representation):

هناك بعض أنظمة الحاسوب التي تقوم بتخزين الأعداد الصحيحة على أساس عشري باستخدام الشيفرة العشرية المكونة من أربع خانات، والتي تعرف بالاسم (Binary Coded Decimal: BCD).

ووفقاً لهذا النوع من التخزين فإن كل جزء من الرقم (منزلة) يمثل بأربع خانات ثنائية، وتمثل الإشارة بأربع خانات أخرى. فإذا أردنا أن نمثل الرقم (9613)، فإن ذلك يمكن أن يتم وفقاً لهذه الطريقة كما يلي (الشكل (2)):

+9613	1001	0110	0001	0011	1100
	↑	↑	↑	↑	↑
	9	6	1	3	±
	↓	↓	↓	↓	↓
-9613	1001	0110	0001	0011	1101

الشكل (2): التمثيل العشري للأعداد الصحيحة (مثال: 1 9613)

2.2 الأعداد الحقيقية (Real)

عزيزي الدارس، الأعداد الحقيقية كما تعلم، هي تلك الأعداد التي تشتمل على فاصلة عشرية. وهي تستخدم لأغراض القياس كقياس الضغط، والمسافات، والأوزان وغير ذلك. وبحكم طبيعة هذه القياسات لا يمكن أن تكون دقيقة بدرجة مطلقة. ولهذا فإن معظم الأجهزة العلمية لا يزيد مستوى دقتها عن سبع أو ثمان خانات عشرية.

وتخزن معظم الحواسيب الأعداد الحقيقية بطريقة تسمح بإجراء العمليات الحسابية بسرعة عالية. ولكن الثمن الذي ندفعه مقابل ذلك عدم الاحتفاظ بالدقة المطلقة. وتمثل الأعداد الحقيقية بطريقة تدعى طريقة النقطة العائمة.

تمثيل النقطة العائمة (floating-point representation):

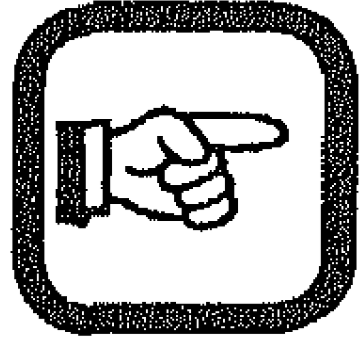
وهذه الطريقة في التخزين هي أكثر أساليب التمثيل شيوعاً، ورغم الاختلافات الموجودة بين الأجهزة في هذا الصدد، فإن هناك نمطاً عاماً مشتركاً يمكن أن نشــير إليه هنا. وهذا النمط يتخذ الشكل التالي (الشكل (3)):

$Sign_M$	$Sign_E$	Exponent الأس	Mantissa القاعدة

الشكل (3): صيغة تمثيل النقطة العائمة

يمثل الرقم الحقيقي بطريقة النقطة العائمة كما يلي:

1. يحول الرقم إلى نظيره في النظام الثنائي.
2. يعبر عن الرقم الثنائي بالصيغة الأسية بتحريك الفاصلة إلى أقصى اليسار والضرب في الأس المناسب.
3. يمثل الأس والقاعدة في المكان المخصص لهما، والمثال التالي يوضح هذه الخطوات:



مثال (1)

مثل الرقم 5.75- بطريقة النقطة العائمة.

الحل:

1. نحول الرقم 5.75 (بتجاهل الإشارة) إلى النظام الثنائي فنحصل على $(101.11)_2$
2. نعبر عن الرقم الثنائي بالصيغة الأسية. لاحظ أن:

$$101.11 = 0.10111 \text{ X } 2^3$$

وتدعى القيمة 0.10111 بالقاعدة (Mantissa) والقيمة 11 بالأس (Exponent).

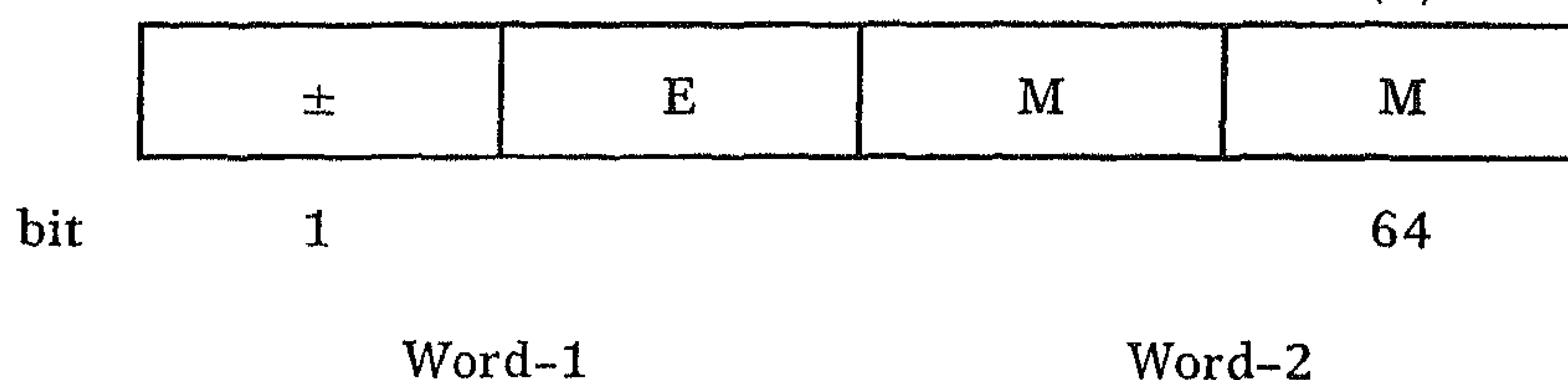
3. تخزن كل من قيمة القاعدة والأس في المكان المخصص لهما، وكذلك إشارتي القاعدة والأس كما يظهر في الشكل (4).

إشارة القاعدة	إشارة الأس	الأس	القاعدة
1	0	11	10111

الشكل (4): قيمة وإشارتي القاعدة والأس

والفائدة الأساسية لهذا النوع من التمثيل أنه يتيح لنا أن نخزن في ذاكرة الحاسوب أعداداً ذات قيمة كبيرة مما لا يسهل تخزينه بالصورة الاعتيادية في موضع واحد من مواضع الذاكرة، وفي معظم الحواسيب يتم تخزين الأرقام الممثلة بهذه الصورة في موضع تخزيني واحد. ولكن في حالة التطبيقات العلمية فإن موضعاً واحداً لا يكفي للحصول على درجة كافية من الدقة.

ولهذا فإن معظم أجهزة الحواسيب لديها ما يسمى بالدقة المضاعفة (double precision)، وتستعمل لهذه الغاية موضعين لتخزين الجزء الكسري، كما هو مبين في الشكل (5).



الشكل (5): صيغة تمثيل الدقة المضاعفة

3.2 الرموز (Characters)

نعني بالرموز الحروف والأعداد من صفر إلى تسعة والعلامات الخاصة. وعلى الرغم من أن الرموز هي أحد الأنماط البيانية البسيطة، إلا أن أهميتها لا ترقى إلى المستوى الخاص بالأعداد الصحيحة والأعداد الحقيقية، وذلك لأن القليل جداً من المسائل والمواقف يمكن التعبير عنها برمز واحد. ولهذا فإن لغة مثل PL/I لا تتضمن هذا النمط في تصميمها، واستعاضت عنه بسلاسل الرموز (strings).

أما من حيث التخزين، فيتم تمثيل الرموز كسياق ثابت الطول من الخانات الثنائية (bits) التي تستمد من الشيفرة المستخدمة مثل ASCII وEBCDIC.

4.2 القيم المنطقية (Logical)

تكون القيم المنطقية، أو البولينية (Boolean) كما نسميها في كثير من الأحوال، إمّا: TRUE أو FALSE بصرف النظر عن لغات البرمجة المختلفة التي تستعملها. وأهم العمليات التي تجري عليها هي عملية الإسناد والعمليات البولينية الثلاثة المعروفة: NOT, OR, AND.

أما عن تخزينها، فإن أسلوب تمثيلها يعتمد على مترجم اللغة والجهاز المستعمل. وبصفة عامة فإن هناك طرقاً ثلاث لتمثيل القيم المنطقية، وهي:

أ. استخدام خانة ثنائية واحدة (bit): وفي هذه الحالة يعبر عن القيمة الإيجابية بالواحد (TRUE = 1) والقيمة السلبية بالصفر (FALSE = 0) وغالباً ما يتم اختيار الخانة الثنائية الخاصة بالإشارة لتخزين القيمة المنطقية بهذا الشكل.

ب. استخدام الموضع التخزيني بكامله (WORD): وفي هذه الحالة يتم تخزين القيمة صفر في الموضع بكامله للتعبير عن FALSE، وتعبئة الموضع بكامله بالقيمة واحد للتعبير عن TRUE.

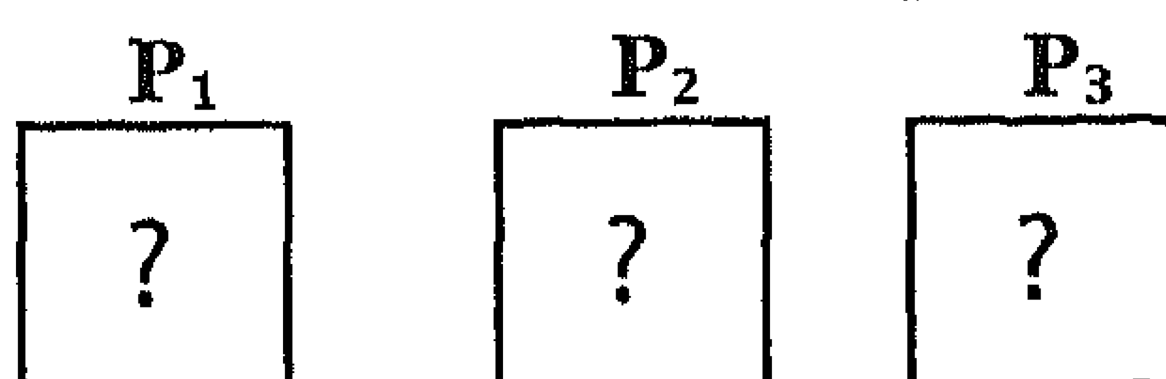
ج. استخدام بايت واحدة من الموضع (word) (لأن الموضع (word) قد يحوي أكثر من بايت): وفي هذه الحالة يتم تخزين القيمة المنطقية في أصغر وحدة معنونة، وبذلك نتجنب مشكلات الطريقة الأولى ونستثمر مواضع الذاكرة بشكل أفضل من الطريقة الثانية.

5.2 مؤشرات الربط (Pointers)

يعدّ مؤشر الربط بحد ذاته نمطاً بيانياً بسيطاً، ومهمته الإحالة إلى موضع آخر في ذاكرة الحاسوب، حيث تخزن إحدى القيم البيانية أو مجموعة منها. ومن هنا فإن محتوى مؤشر الربط هو عنوان ذلك الموضع. وقد يكون مؤشر الربط مستقلاً بذاته وقد يكون جزءاً من مركب بياني آخر، كأن يكون حقلاً في سجل. دعنا نرى طبيعة عمل المؤشر، من خلال التعريف التالي في لغة C/C++:

```
float *P1, *P2, *P3;
```

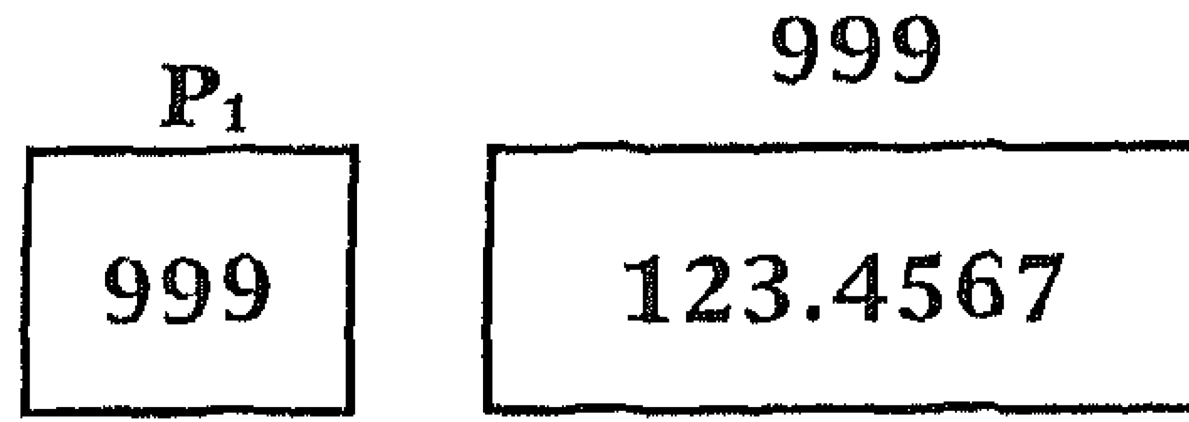
إن معنى هذا التعريف هو أن لدينا ثلاثة متغيرات هي: P1, P2, P3، وكل منها يشير إلى قيمة عددية حقيقية. وعلى هذا الأساس فإن المترجم يقوم بحجز ثلاثة أماكن في الذاكرة لهذه المتغيرات، كما في الشكل (6 – أ):



الشكل (6 – أ): مؤشرات الربط

والآن افترض أن لدينا جملة الإسناد التالية داخل البرنامج: `*P1=123.4567;`

فإن ذلك يعني تخزين القيمة (123.4567) في موضع معين في الذاكرة وإسناد عنوان هذا الموضع إلى المتغير.P1 فإذا افترضنا أن هذا العنوان هو (999) فإن الموضع بعد تنفيذ جملة الإسناد سيصبح كما في الشكل (-6ب):



الشكل (6 - ب)

ولعلك، عزيزي الدارس، قد وصلت إلى النتيجة البسيطة: بأن هناك إمكانية لأن نسند هذه القيمة إلى متغيرات أخرى مثل P2 وP3. وكما هو واضح فإن المساحة التي يحتلها أي مؤشر في الذاكرة أقل من المساحة التي تحتلها القيمة (123.4567).

وفي أغلب الحواسيب يخزن المؤشر في موضع واحد من مواضع الذاكرة أو في نصف موضع. أما القيمة المخزنة في المؤشر فيمكن أن تكون العنوان المطلق (absolute address) للموضع الذي يشير إليه وذلك على النحو الوارد في المتغير «P1» حيث الرقم (999) هو عنوان الموضع الذي يشير إليه «P1» أو العنوان النسبي (relative address) لهذا الموضع ضمن منطقة معينة من الذاكرة تم تخصيصها لهذا الغرض وذلك على نحو مماثل للأرقام التي نستخدمها في الإشارة إلى عناصر المصفوفة.

أسئلة التقويم الذاتي (1)



1 - أجب بنعم أو لا على العبارات الآتية:

1. عندما نقوم بتنظيم البيانات في مجموعات، وجداول، وقوائم، وسجلات، وملفات، وغير ذلك من أساليب التنظيم فإن هذه العملية يطلق عليها تركيب البيانات.
2. تعتبر الحروف والأرقام والسجلات والمصفوفات من الأنماط البيانية البسيطة في لغات الحاسوب.
3. يمكن التعامل مع كل عنصر في أي مركب بياني على حدة كما لو كان متغيراً مستقلاً كبقية المتغيرات.

4. طريقة تنظيم العناصر في أي مركب بياني لا تؤثر كثيراً على طريقة الوصول إلى العناصر الفردية.
5. المصطلح تراكيب بيانية بدائية يتضمن الأعداد الصحيحة، والأعداد الحقيقية، والقيم المنطقية، والرموز، ومؤشرات الربط.
6. تخزن الأعداد الصحيحة في ذاكرة الحاسوب وفق طريقة المكمل الأول (1's complement).
7. باستخدام طريقة التمثيل بالنقطة الثابتة نستطيع أن نخزن في الموضع الواحد من مواضع الذاكرة قيماً أكبر مما يتاح بطريقة النقطة العائمة.

2 - أكمل الفراغات في العبارات الآتية:

1. القيمة التي تخزن في المتغير المعرف على أنه مؤشر ربط هي............
2. مؤشر الربط يخزن عادة في............
3. هناك ثلاثة أساليب لتمثيل القيم المنطقية في الذاكرة هي............
4. يعتمد تمثيل الحروف الهجائية في ذاكرة الحاسوب على............
5. يعبر عن الصيغة العامة لتمثيل الأرقام الحقيقية باستخدام أسلوب النقطة العائمة من خلال العلاقة التالية............
6. عندما نمثل كل خانة من أحد الأرقام الصحيحة بأربع خانات ثنائية (bits) فإن هذا النوع من التمثيل يسمى............
7. الفرق بين المصطلح «تركيب بياني» (data structure) والمصطلح «تركيب البيانات» هو أن الأول يشير إلى ....
.... بينما يشير الثاني إلى ....
....
8. توفر بعض اللغات أنماطاً بيانية مركبة ضمن تصميمها، ومن أمثلة هذه الأنماط ....

3. التركيب المنطقي والفيزيائي للبيانات (Logical and Physical Structure of Data)

أشرنا في بداية حديثنا في هذه الوحدة إلى أن دراسة تركيب البيانات تهتم بالكيفية التي يتم بها ترجمة التصور المنطقي لمجموعة من البيانات إلى تنظيم اختزاني خاص يتفق وطبيعة تصميم ذاكرة الحاسوب ويتيح لنا إمكانية الوصول إلى العناصر المختلفة لهذه البيانات. ويجدر بنا أن نسأل الآن: هل هناك اختلاف بين التصور المنطقي لدى المبرمج والتركيب المادي أو الفيزيائي للذاكرة حتى نقوم بعملية ترجمة تصور الواحد للآخر؟ والإجابة البسيطة على هذا السؤال هي بالإيجاب. ولكن قبل أن نبين وجه الاختلاف، دعنا، عزيزي الدارس، نبين ما نقصده بالتركيبين المنطقي والفيزيائي.

لا بد وأنك اطلعت، في حياتك العملية اليومية، على عمل بعض المهندسين الذين يشرفون على شق الطرق وبناء الجسور والمباني، أو على الأقل أنك رأيت فِرَق العمل في أثناء التنفيذ. ولا بد أنك رأيت بعضهم يحملون المخططات ويقومون بإجراء القياسات حتى في أبسط أعمال البناء. ماذا يفعل هؤلاء؟ ولم هذه القياسات والمخططات؟ وكيف يتم التنسيق بين عمل المهندسين والفنيين والعمال؟ وما علاقة صاحب العمل بكل هذا؟ كل هذه الأسئلة لها علاقة بمسألة التركيب المنطقي والتركيب المادي التي نحن بصددها.

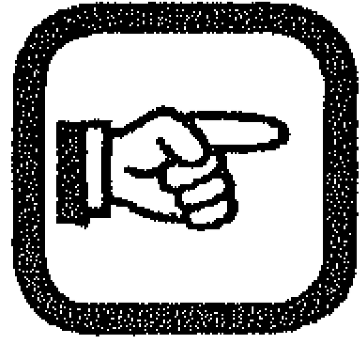


مثال (2)

تأمل في بناء أحد ملاعب كرة القدم، في إحدى المدن الرياضية. فمن وجهة نظر المهندس المعماري، مثل هذا البناء له أبعاد ومسافات خاصة، وفيه جوانب فنية وجمالية معينة ينبغي أن تتوافر. أما بالنسبة للمهندس الإنشائي، فإن هذا الملعب، بأجزائه المختلفة، لا بد أن يتكون من مواد خاصة وأن تكون له ركائز معينة. فعلاقة هذه المواد ببعضها وعدد الركائز وسمكها وعلاقتها ببعضها ومدى مقاومة هذه المواد والركائز والمكونات الأخرى للمبنى هي ما يشغل ذهن المهندس الإنشائي. أما بالنسبة للرياضي والمتفرج، فإن هذا البناء له وظيفة معينة يؤديها، ولذلك فهو ينظر إلى مدى التزام الملعب بالمبادئ الدولية المتعارف عليها، وعدد الأماكن المتوفرة للمتفرجين، وتوزيع بوابات الدخول والخروج وغير ذلك من المسائل التي تهم اللاعب والمتفرج. وفي النهاية فإن ترجمة هذه التصورات على أرض الواقع، إلى شيء ملموس، هي من مهام القائم على التنفيذ، والذي ينظر إلى هذا البناء على أساس أنه مجموعة من الموارد والإمكانات البشرية والمادية.

فكما ترى في هذا المثال، كل شخص من هؤلاء له تصوره الخاص لهذا البناء. وما أوردنا هذا المثال إلا لنبين لك، عزيزي الدارس، أن الشيء الواحد قد ينظر إليه من زوايا مختلفة. وهذا هو حال البيانات؛ فنحن في هذا الصدد نتعامل مع فريقين أساسيين: الأول هو المبرمج، والذي ينظر إلى البيانات من زاوية خاصة، والفريق الثاني هو الحاسوب الذي «يدرك» معنى البيانات بصورة أخرى مختلفة. فنظرة المبرمج إلى البيانات هي ما نسميه بالتصور المنطقي، وهي نظرة نابعة من الوظيفة التي تؤديها هذه البيانات، و«نظرة» الجهاز إليها هي ما نسميه بالتصور الفيزيائي، وهي نظرة نابعة من طبيعة تكوين الجهاز وطريقة تعامله مع البيانات من حيث التخزين.

وحقيقة الأمر أننا إذا تجاوزنا اختلافات تمثيل البيانات في الذاكرة تبعاً لنوع البيانات، وهي المسألة التي كانت محور اهتمام القسم السابق في هذه الوحدة، فإن نظرة الحاسوب إلى البيانات تكاد تكون واحدة مهما اختلفت أساليب التنظيم ومستوياته. فالذاكرة مكونة من مجموعة من المواضع المعنونة تتبع بعضها بعضاً على نسق واحد. فهي أشبه ما تكون بمصفوفة ذات بعد واحد، أو هي كذلك فعلاً. ومن هنا يأتي الاختلاف بين النظرة المنطقية للمبرمج والتصور الفيزيائي للجهاز. ولكي نوضح الفرق بين وجهتي النظر، دعنا، عزيزي الدارس، نتأمل الأمثلة التالية:



مثال (3)

لا بد أنك استخدمت أحد أجهزة الهاتف الذي يعمل عن طريق الضغط على مجموعة من المفاتيح، وليس بالقرص، والتي يحمل كل مفتاح منها رقماً واحداً، كما في الشكل (7). فبالنسبة للمبرمج يمكن التعبير عن لوحة المفاتيح الخاصة بهذا النوع من أجهزة الهاتف بمصفوفة ذات بعدين، وعلى هذا الأساس فإن الشكل (8) يمثل تصور المبرمج المنطقي لهذه المسألة، والذي يمكن أن يعبر عنه المبرمج بلغة سي++ كما يلي:

1	2	3
4	5	6
7	8	9
*	0	#

الشكل (7): لوحة مفاتيح أرقام الهاتف

N→	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	*	0	#

char A[4][3];

الشكل (8): التركيب المنطقي في ذهن المبرمج للمصفوفة A

هذا هو الأمر بالنسبة للمبرمج. ولكن ماذا عن الجهاز أو ما نسميه بالتركيب الفيزيائي؟ فالبيانات المذكورة أعلاه ستمثل في الذاكرة تباعاً عنصراً بعد عنصر، كما في الشكل (9).

0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	*	0	#

الشكل (9): التركيب الفيزيائي في داخل الذاكرة للمصفوفة A

من الواضح أن هناك فرقاً بين التصورين: المنطقي والفيزيائي. وهذا الفرق يقودنا إلى التساؤل عن كيفية الوصول إلى أحد العناصر. فبناء على التعريف المعطى أعلاه وبناء على الإمكانات التي توفرها اللغة، إذا أراد المبرمج أن يصل إلى محتويات أحد العناصر، فما عليه إلا أن يستخدم: A [I][J] ولا يحتاج إلى معرفة التفاصيل الداخلية لعملية الوصول. أما بالنسبة للجهاز فتتم عملية الوصول بإجراء عملية حسابية خاصة نحدد من خلالها العنوان النسبي للعنصر المطلوب بالنظر إلى عنوان البداية (base)، وهو عنوان العنصر الأول، كمـا يلي:

$$a\,(A[I][J]) = base + size\,(offset)$$
$$Offset = (I - LB)\,n + (J - LB)$$

حيث:

«base»: العنوان المطلق للبداية.

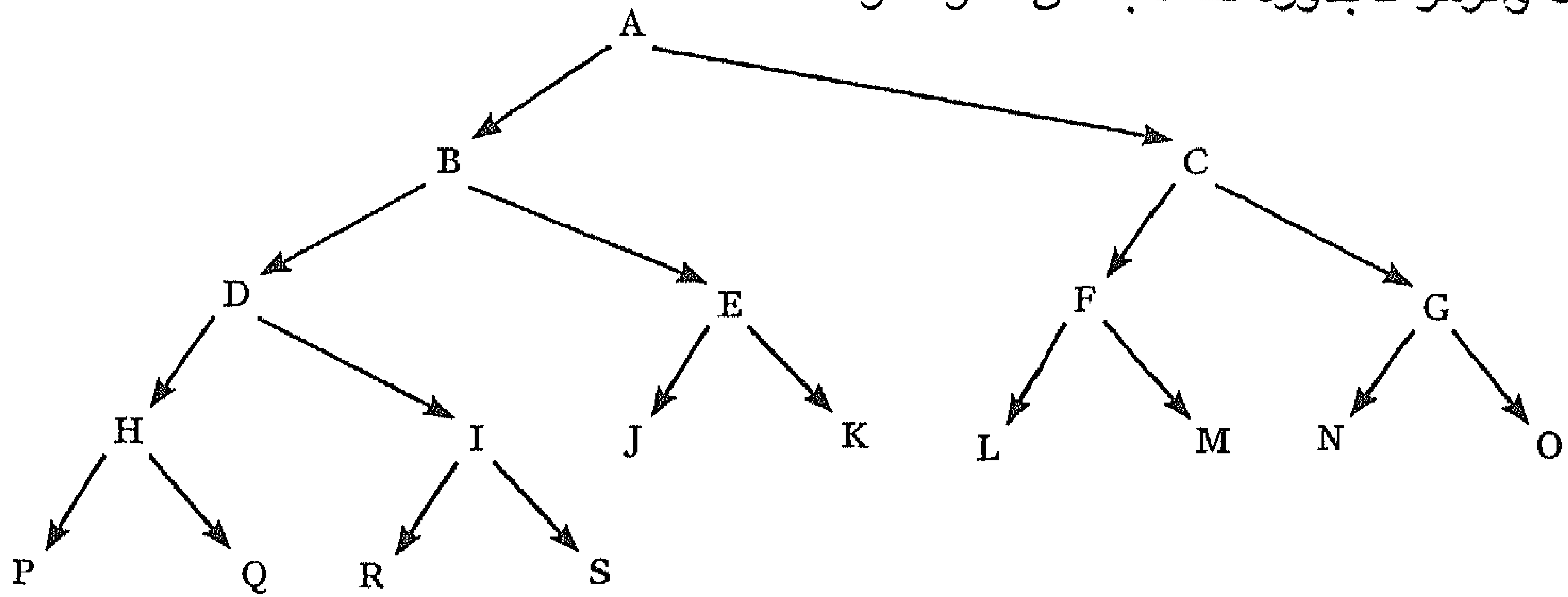
«size»: عدد المواضع التي يحتلها العنصر الواحد.

«LB»: الحد الأدنى لكل بعد في المصفوفة.

وعلى هذا الأساس، فإن جهد المبرمج سينصرف إلى حل المشكلة نفسها بدلاً من الاهتمام بالتفاصيل الداخلية والتركيب الفيزيائي للبيانات في داخل الذاكرة.

مثال (4)

يختلف التركيب المنطقي للبيانات باختلاف المشكلة التي يتعامل معها المبرمج. فقد رأينا في المثال (3) السابق أن المبرمج قد عبر عن المسألة بمصفوفة ذات بعدين، ولكن تأمل لو كانت لدينا علاقة من النوع الذي يعبر عن سيد ومرؤوس أو والد وولد (أي علاقة نسب عائلي)، وإن علاقة التابع بمتبوعه تتمثل بصورة ثنائية، أي أن لكل متبوع تابعين اثنين، فإن مثل هذه العلاقة يمكن أن يتصورها المبرمج من الناحية المنطقية على شكل شجيرة كما هو موضح في الشكل (10). ففي هذا الشكل نجد أن كل متبوع (parent) له تابعان (left child, right child) فالرمز A متبوع بالرمزين B و C والرمز B بدوره له تابعان D وE. وهكذا.



الشكل (10): تمثيل منطقي لعلاقة هرمية كما يراها المبرمج

وفي المقابل فإن التعبير عن هذه العلاقة في ذاكرة الحاسوب لا بد وأن يتخذ شكلاً آخر ينسجم مع طبيعة التخزين فيها، وبعبارة أخرى، فإننا مضطرون لتخزين هذه العناصر بطريقة تتابعية في ذاكرة الحاسوب مع محاولة إيجاد الوسيلة المناسبة للمحافظة على العلاقة الهرمية. ومن هنا يتم تخزين هذه العناصر على النحو الموضح في الشكل (11).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	...

الشكل (11): تمثيل فيزيائي للعلاقة الهرمية الموضحة في الشكل (10)

ومن خلال هذا التمثيل الفيزيائي في الذاكرة يمكن أن نحافظ على العلاقة المنطقية المبينة أعلاه عن طريق إجراء بعض الحسابات. فإذا أردنا أن نعرف العنصرين التابعين للرمز H على سبيل المثال فإننا نطبق المعادلتين التاليتين لإيجاد موضع كل منهما:

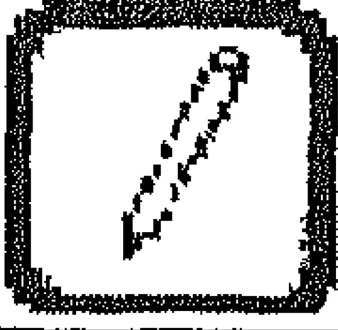
$$[\text{left child} = \text{parent} * 2] = 8 * 2 = 16$$

$$[\text{right child} = \text{parent} * 2 + 1] = 8 * 2 + 1 = 17$$

حيث يرمز الرقم 8 إلى موضع H في ذاكرة الحاسوب، وإذا أردنا أن نعرف المتبوع بالنسبة للعنصر E مثلاً، نحصل على ذلك من خلال المعادلة التالية:

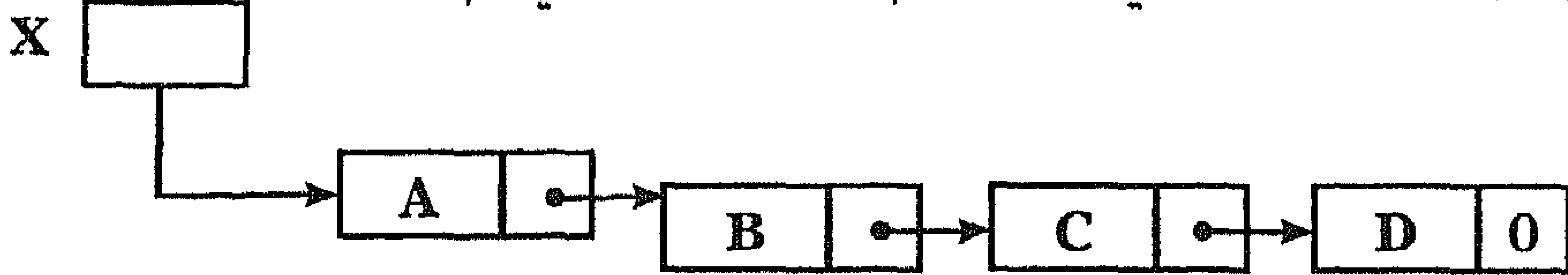
$$[\text{parent} = \text{INT}\ (\text{child}/2)] = 5/2 = 2$$

أي أن العنصر الذي يتبع له «E»، في هذا المثال، هو العنصر الوارد في الموضع «2» وهو الرمز «B».



تدريب (1)

تصور أنك تستخدم لغة لا تشتمل على نمط بياني خاص بمؤشرات الربط (Pointers) كما هو الحال في لغة الفورتران، فماذا يمثل الرسم المبين في الشكل (12): التركيب المنطقي للبيانات، أم التركيب الفيزيائي، أم كليهما معاً؟



الشكل (12): قائمة خطية بعدد من العناصر بالاسم X



أسئلة التقويم الذاتي (2)

أجب بنعم أو لا على العبارات الآتية:

1. ليس هناك اختلاف بين التركيب المنطقي للبيانات والتركيب الفيزيائي لها.
2. إذا كان التركيب الفيزيائي للبيانات مدعوماً من الجهاز فإن المبرمج لا يحتاج لمعرفة التفاصيل الداخلية الخاصة بهذا التركيب، كما هو الحال في المصفوفات.
3. يمكن النظر إلى ذاكرة الحاسوب على أنها مصفوفة كبيرة ذات بعد واحد.
4. التركيب المنطقي للبيانات لا يختلف باختلاف المشكلة التي يتعامل معها المبرمج.
5. جرياً على مبدأ تخزين البيانات بطريقة تتابعية في الذاكرة، فإن الحقول المختلفة المكونة لأحد لسجلات تخزن جنباً إلى جنب.
6. إن طريقة تخزين التراكيب البيانية في الذاكرة تختلف باختلاف التركيب المنطقي للبيانات.

4. تراكيب البيانات المختلفة وأهميتها والعمليات المنفذة عليها (Examples, Importance, and Operations)

1.4 أهمية تركيب البيانات

لقد قمنا فيما سبق، عزيزي الدارس، بتحديد مفهوم تركيب البيانات، وفرقنا بين مسألة تركيب البيانات كعملية وكدراسة لها مناهجها وأساليبها، والتراكيب البيانية كأساليب تنظيمية للبيانات وهي الناتج الفعلي لعملية التركيب. وهنا، في هذا القسم من الوحدة الأولى، نحاول أن نبين أهمية وجود مثل هذه التراكيب البيانية المختلفة، ونقدم نماذج منها تمهيداً للحديث المفصل عنها في الوحدات القادمة، والعمليات التي يمكن تنفيذها عليها بصفة عامة.

إن هناك الكثير من الأشياء والأفكار في الحياة اليومية والممارسة العملية نأخذها كمسلمات أحياناً دون أن نناقش الفلسفة التي تقوم عليها والمسوغات التي أوجدتها. ولربما كانت عملية تركيب البيانات من بين هذه المسائل التي نسلم بها ونقوم بتنفيذها دون أن نكلف أنفسنا عناء البحث عن أهميتها. فالإنسان بحكم طبيعته الذهنية، وبحكم طبيعة حياته العملية يقوم من تلقاء نفسه بفرض نظام معين على البيانات التي يستقبلها من محيطه. فالمدرس ينظم علامات طلبته على نحو يساعده على إيجاد علامات أي طالب دون جهد كبير، والمحامي ينظم قضاياه ومواعيده بشكل يساعده على إيجاد تسلسل مناسب لأعماله، وكذلك يفعل الطبيب وغيرهم.

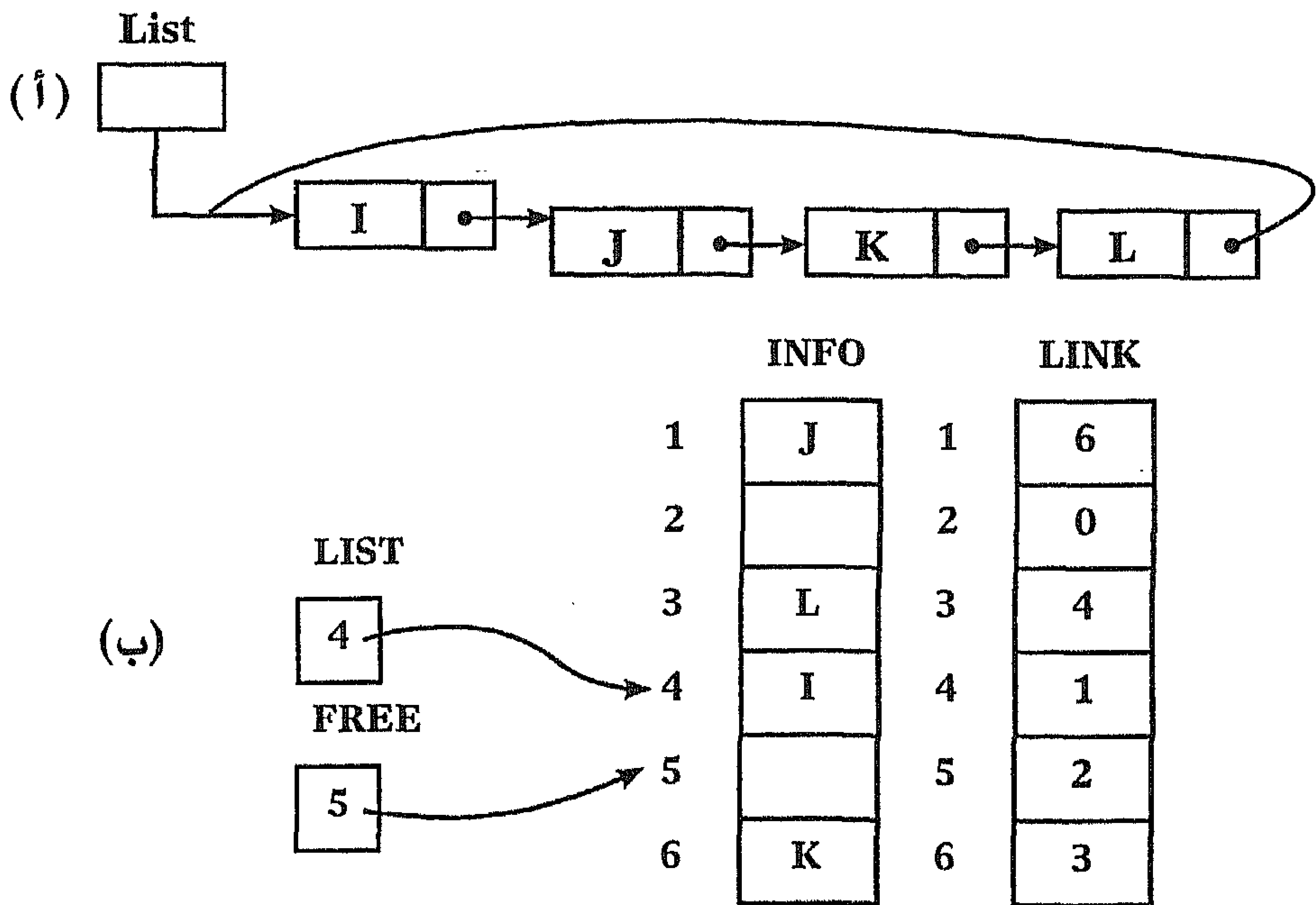
وعندما يتصل الأمر بالمبرمج، فمن المسلمات الأساسية أنه يحتاج لإيجاد تنظيم مناسب للبيانات التي يتعامل معها بحيث يكون هذا التنظيم على درجة عالية من الكفاءة، بخاصة عندما يكون حجم البيانات كبيراً وتكون العلاقات المنطقية بينها متشعبة ومتداخلة. والمسألة تصبح أكثر أهمية عندما نتذكر بأن جهاز الحاسوب يفرض على المبرمج قيوداً كثيرة في التخزين والمعالجة.

إن القيم البيانية التي نتعامل معها تمثل أحداثاً وأشياء في العالم الذي يحيط بنا، وحتى تستطيع هذه القيم البيانية أن تعبر عن هذه الأشياء والأحداث تعبيراً صادقاً فلا بد من إيجاد تنظيم خاص يحكمها. وبالطبع فإن هذا التنظيم مشتق من طبيعة العلاقات المنطقية القائمة بين هذه البيانات.

وعلى هذا الأساس فإن أهمية عملية التركيب تكمن أولاً في حرصنا على أن نعكس العلاقات المنطقية القائمة بين الأشياء التي تمثلها على اختلاف أنواعها. ولكن أهمية التراكيب البيانية لا تقف في الواقع عند هذا الحد. فالمبرمج المبتدئ الذي يتعامل مع عدد محدود من القيم

البيانية لا يهمه ما إذا كان تنظيم هذه القيم مناسباً أو لا. ولذلك فإن مسألة تنظيم هذه القيم على نحو يساعده على استغلال الموارد الحاسوبية بشكل كافٍ لن يشغل ذهنه كثيراً. وفي المقابل فإن الأمر مختلف جداً بالنسبة للمبرمج المتمرس. فهذا الأخير يضع نصب عينيه أن يستثمر الموارد الموجودة أفضل استثمار سواء من حيث الوقت أو من حيث المساحة. ولذلك فإنه يحرص على اختيار التركيب البياني الذي يحقق هذا الهدف.

وبالإضافة إلى ذلك، فإن بعض لغات البرمجة تفرض قيوداً خاصة على المبرمج في مجال تنظيم البيانات وتخزينها ومعالجتها. صحيح أن التصور المنطقي للعلاقة الهرمية بين عناصر البيانات واحد بالنسبة للمبرمج إلا أن لغة مثل لغة الفورتران لا تتيح لك إمكانية التعبير عن مثل هذا التنظيم الهرمي للقيم البيانية على شكل شجيرة إلا من خلال استخدام المصفوفات المتوازية والوصلات أو المؤشرات النسبية (relative pointers) التي يوجدها المبرمج نفسه وليس اللغة. وهذا على خلاف لغة سي/سي++ التي تتيح المجال للمبرمج أن يستحدث الكثير من التراكيب البيانية باستخدام مؤشرات الربط التي توفرها اللغة نفسها. ولتوضيح هذه الفكرة، تأمل، عزيزي الدارس، الشكل (13) الذي يعبر عن تركيب بياني يستخدم الوصلات.



الشكل (13): تمثيل قائمة من العناصر باستخدام المصفوفات المتوازية والمؤشرات النسبية في لغة الفوتران

فكما تلاحظ في هذا الشكل، فإن الجزء العلوي (أ) يمثل التصور المنطقي لقائمة من العناصر يشير كل منها إلى الآخر، أما الجزء السفلي (ب) فهو يمثل طريقة التخزين باستعمال المصفوفتين المتوازيتين INFO وLINK. وكما يتضح من المصفوفة LINK فإن القيمة المخزنة فيها هي قيمة الوصلة. ومن هنا فإن القيمة المقابلة للحرف I هي واحد، بمعنى أن العنصر التالي هو ذلك المخزن في الموضع الأول من المصفوفة INFO، والقيمة المقابلة للحرف J هي الرقم 6، أي بمعنى أن J تشير إلى K المخزنة في الموضع السادس... وهكذا، على نحو يطابق التصور المنطقي المعطى أعلاه. لاحظ أن الهدف من FREE هو الاحتفاظ بقائمة بالأماكن الشاغرة.

وإذا كانت هذه هي الأسباب التي تدعونا إلى الاهتمام بدراسة تركيب البيانات، فالسؤال اذن هو: أي أسس نختار بموجبها من بين التراكيب البيانية المتوفرة لحل مشكلة معينة؟ للإجابة على هذا السؤال نورد فيما يلي مجموعة من العوامل التي تؤثر على الاختيار:

1. حجم البيانات المتصلة بالمشكلة.
2. مدى تكرار حدوث المشكلة والطريقة التي ستستعمل بها البيانات.
3. طبيعة البيانات نفسها، من حيث مدى ثباتها أو مدى التغيير الذي تتعرض له.
4. مساحة الذاكرة التي يتطلبها التركيب البياني لأغراض التخزين.
5. الوقت اللازم لاسترجاع أحد عناصر البيانات.
6. مدى سهولة أو صعوبة ترجمة التركيب البياني إلى جمل برمجية بإحدى لغات البرمجة.

فعلى سبيل المثال، إذا كانت لدينا مشكلة نريد حلها بسرعة، فلن نأبه في هذه الحالة إلى مدى كفاءة التركيب البياني الذي تم اختياره من حيث استغلال الموارد. فالهدف هنا هو الحصول على حل سريع بصرف النظر عن نوعية البرنامج المكتوب. وفي المقابل يمكن أن نكتب برنامجاً معقداً يستغرق منا أشهراً وربما سنوات ويوفر لنا إمكانات على درجة عالية من الكفاءة. ولكن كما ترى، في هذه الحالة لن ينتظر الحل كل هذه المدة. وعلى أي حال، ليس هناك قاعدة محددة يمكن الالتزام بها في الاختبار. فهناك دائماً أشياء على حساب أشياء أخرى، ونلجأ في كثير من الأحوال إلى الحلول الوسط (trade offs) وإعطاء الأولوية لشيء على حساب شيء آخر.

2.4 نماذج من التراكيب البيانية

ثمة أنواع مختلفة من تراكيب البيانات. وإذا عدت، عزيزي الدارس، إلى الشكل (1) في بداية هذه الوحدة، فإنك ستجد تصنيفاً مبسطاً لهذه التراكيب. وقرار المبرمج في اختيار أحد هذه التراكيب البيانية دون غيره يعتمد على عاملين أساسيين: أولهما، هو أن يعكس التركيب البياني العلاقات الحقيقية بين البيانات والأشياء التي تمثلها في عالم الواقع؛ وثانيهما هو أن يكون التركيب البياني بسيطاً بدرجة كافية، وفي الوقت نفسه على درجة مناسبة من الكفاءة لمعالجة البيانات وقت الحاجة. وفيما يلي نعرض أهم التراكيب البيانية المستعملة بكثرة:

أ - المصفوفات (Arrays):

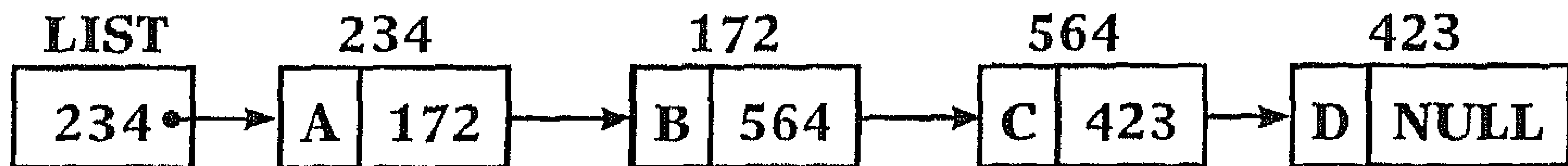
ليست هذه هي بالمناسبة المهمة الأولى التي تسمع بها عن المصفوفات. فلقد درست عنها ومارستها من خلال مقررات البرمجة ومقررات أخرى.

وتعد المصفوفة الخطية (linear) أو ذات البعد الواحد أبسط أنواع المصفوفات. ويتم تمثيلها في الذاكرة على شكل سلسلة من المواضع المتجاورة، عددها مساوٍ لعدد عناصر المصفوفة. أما المصفوفة ذات البعدين فإنها كسابقتها تتكون من مجموعةٍ من العناصر المتجانسة، إلا أن طريقة تخزينها تنطوي على بعض المعاملة الخاصة كما لاحظنا في المثال (3) عند الحديث عن التركيب المنطقي والفيزيائي للبيانات. وبصفة عامة يتم تخزين بيانات المصفوفة الثنائية في أماكن متجاورة، إلا أن هناك اختلافاً بين لغات الحاسوب من حيث تسلسل العناصر، فبينما تسير لغة سي++، مثلاً، على طريقة الصفوف تتعامل لغة الفورتران مع القيم حسب تسلسلها في الأعمدة. كما يتم التعامل مع المصفوفات المتعددة الأبعاد بالأسلوب نفسه. وفي الوحدة الثانية سنرى الفرق بين طريقة التمثيل الأفقي (حسب الصفوف) وطريقة التمثيل الطولي (حسب الأعمدة).

ب - القوائم المتصلة (Linked lists):

لاحظنا أن العلاقة الخطية في المصفوفات ذات البعد الواحد تنعكس من خلال التجاور الفيزيائي لعناصر المصفوفة. وبالإضافة إلى ذلك توجد طريقة أخرى للتعبير عن هذه العلاقة الخطية وذلك بتخزين معلومات في كل عنصر عن موضع العنصر التالي في السياق الخطي المقصود. وإذا عدت إلى الشكل (13) في بداية هذا القسم، فإنك ستجد هذا النوع من التمثيل الخطي. ففي هذا الشكل استخدمنا مصفوفتين: إحداهما للقيم التي تصل العناصر ببعضها، والأخرى للبيانات الحقيقية نفسها.

هذه الطريقة، المبينة في شكل (13)، للتعبير عن العلاقة تصلح، بصفة خاصة، في اللغات التي لا تشتمل على نمط بياني متميز لغايات الإشارة فيما يصطلح عليه باسم نمط مؤشر (pointer type). وفي المقابل هناك لغات مثل سي++ تتضمن مثل هذا النمط، وبذلك تتوفر للمبرمج إمكانات بناء قوائم خطية يشار إليها باسم القوائم المتصلة. والفرق الأساسي بين بناء القوائم المتصلة باستخدام مؤشرات الربط التي توفرها اللغة وبين طريقة المصفوفات المتوازية ليس فقط في نوع المعلومات الإشارية المستخدمة في الدلالة على مكان العنصر التالي، بل أيضاً في طريقة حجز الأماكن في الذاكرة ومدى التقارب بين هذه العناصر من الناحية الفيزيائية. فباستخدام مؤشرات الربط الخاصة باللغة لا تدعو الضرورة إلى وجود تقارب مكاني بين العناصر، بل يتم حجز الأماكن للعناصر المختلفة حسب توفرها. والشكل (14) يوضح فكرة القوائم المتصلة باستخدام مؤشرات الربط المتوفرة في لغة سي++.



الشكل (14): قائمة متصلة خطية

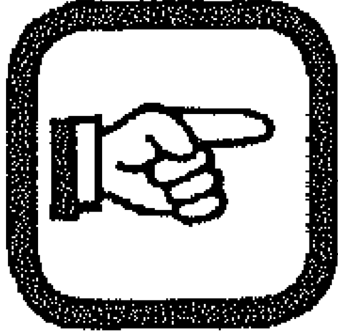
فكما يتضح من هذا الشكل لدينا مؤشر يشير إلى العنصر الأول في القائمة المتصلة. فإذا افترضنا أن مواضع الذاكرة مرقمة بالتسلسل من 0 إلى 999، فإن القيمة المخزنة في المتغير LIST هي العنوان المطلق للعنصر الأول، والذي يشير بدوره للعنصر الثاني من خلال تخزين عنوانه المطلق وهو 172... وهكذا حتى نهاية القائمة، حيث نجد أن قيمة المؤشر في العنصر الأخير هي NULL، بمعنى أنه لا يشير إلى أي عنصر آخر.

3.4 العمليات الأساسية التي تتم على التراكيب البيانية (DATA STRUCTURE OPERATIONS)

إن عملية تركيب البيانات عملية تنظيمية وتخزينية لقيم بيانية فعلية. ولذلك، بمجرد أن تتم هذه العملية تصبح هذه البيانات جاهزة للاسترجاع والمعالجة. وفي الحقيقة أن اختيار أحد التراكيب البيانية يعتمد كثيراً على مدى الحاجة إلى التعامل مع البيانات المعنية وطبيعة هذا التعامل والعمليات التي تجريها على البيانات. وفيما يلي نستعرض، عزيزي الدارس، أهم العمليات التي تتم على التراكيب البيانية:

أ - الاستقصاء (Searching):

يقال في اللغة «استقصى الأمر وتقصاه»: بلغ غايته في البحث عنه. والمفهوم الاصطلاحي لا يختلف كثيراً عن هذا المعنى اللغوي، فعملية الاستقصاء هي عملية البحث عن البيانات. وقد تقتصر العملية على عنصر واحد محدد، وقد تتناول عدداً من العناصر التي تستوفي شرطاً معنياً.

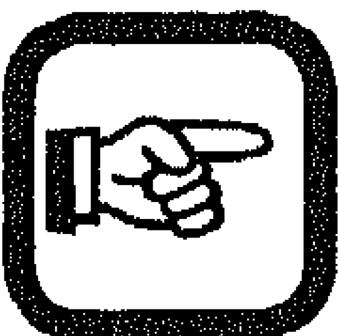


مثال (5)

افترض أن لدينا مصفوفة مكوّنة من عشرة عناصر، وكل عنصر فيها يمثل سجلاً لأحد الطلبة يتضمن رقمه واسمه وعلاماته في أحد المقررات. ويرغب المدرّس في معرفة علامة الطالب (س) في الامتحان النهائي لهذا المقرر. ففي هذه الحالة نقوم بعملية استقصاء للعنصر المطلوب وهو (س) في المصفوفة المعنية بناء على قيمة خاصة تسمى المفتاح Key. وفي هذه الحالة إما أن تكون هذه القيمة الخاصة اسم الطالب أو رقمه. أو افترض أن المدرس يرغب في معرفة أسماء جميع الطلبة الذين حصلوا على علامة (90) أو أكثر في هذا الامتحان. ففي هذه الحالة نقوم باستقصاء كل الحالات التي ينطبق عليها هذا الشرط.

ب - الاستعراض (Traversal):

يقال في اللغة «استعرض المسائل»: بحثها جميعاً واحدة تلو الأخرى. ونقول اصطلاحاً «استعرضنا البيانات»: قمنا بمعالجتها عنصراً بعد آخر على التوالي. وكما هو الحال بالنسبة لعملية الاستقصاء، فإن عملية الاستعراض تنطوي على الوصول إلى العناصر ولكن المعني بالوصول هنا هو زيارة كل عنصر من العناصر التي تحتوي على قيم بيانية مرة واحدة. ومن الحالات التي تقتضي القيام بمثل هذه العملية: طباعة محتويات أحد التراكيب البيانية، والقيام بتعداد القيم البيانية المخزنة، والبحث عن أصغر أو أكبر قيمة.



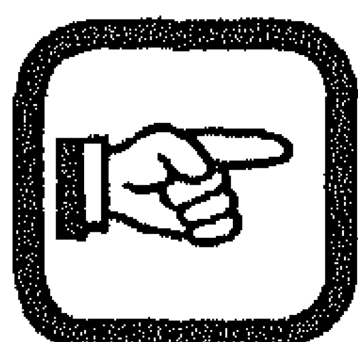
مثال (6)

في ضوء المعلومات المعطاة في المثال (5) السابق ذكره، افترض أن المدرس يرغب في معرفة أعلى أو أدنى علامة حصل عليها أحد الطلبة في الامتحان النهائي. فمعنى ذلك

أننا لا بد أن نزور جميع العناصر حتى نقرر أي العلامات أكبر أو أصغر. وفي كل مرة لا بد أن نجري عملية المقارنة بين أكبر أو أصغر قيمة وصلنا إليها بالقيمة التالية حتى نهاية قائمة الطلبة.

ج - الفرز (Sorting):

يقال في اللغة «فرز الشيء فرزاً»: ميّزه ونحّاه. ونقول اصطلاحاً «فرز البيانات»: رتبها بطريقة معينة. وحقيقة الأمر أن عملية الفرز هي جزء من عملية الترتيب وليس العملية بأكملها. فمعنى الفرز في مجال البيانات لا يختلف كثيراً عن المعنى اللغوي، وهو إعطاء شيء مميز لقيمة بيانية أو مجموعة من البيانات المنظمة على شكل سجل كأن تكون شيفرة خاصة أو رمزاً مميزاً (code). وفي ضوء هذه الشيفرة أو هذا الرمز تتم عملية الترتيب النهائية. وعلى أي حال، فالمعنى الذي نقصده في السياق هو القيام بعملية الترتيب حتى نضع كل قيمة بيانية أو سجل في وضعه الصحيح تبعاً للسياق الترتيبي المعتمد. وهذا السياق قد يكون هجائياً أو رقمياً أو غير ذلك، مما يتم اعتماده في ترتيب البيانات.



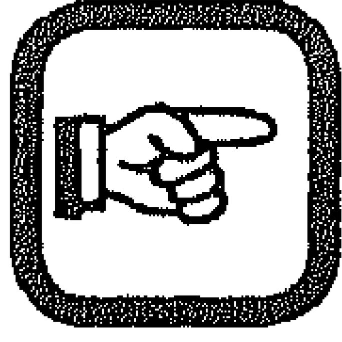
مثال (7)

لم نشر في مثال (6) السابق ما إذا كانت المصفوفة التي تضم سجلّات الطلبة منظمة وفق أي سياق ترتيبي. والآن دعنا نفترض أنها غير مرتبة، ونريد الآن أن نرتبها وفق القيمة المخزنة في أحد الحقول (المفتاح) حتى نسهل عملية الوصول إلى السجلات المطلوبة. وهنا الحقلان اللذان يصلحان لهذه الغاية هما الرقم الجامعي للطالب واسمه، فكلاهما يصلح لأن يكون مفتاحاً للوصول. ومعنى ذلك إذا اخترنا رقم الطالب، ستجري عملية الفرز بناء على ذلك وستكون النتيجة أن لدينا مصفوفة مرتبة تصاعدياً وفقاً للأرقام. وإذا اخترنا الاسم، فتكون المصفوفة بعد عملية الفرز مرتبة وفقاً للحروف الهجائية.

د - الدمج (Merging):

يقال في اللغة «دمج الشيء في الشيء»: دخل واستحكم فيه. ويقال أيضاً «ضم الشيء إلى الشيء»: أضافه إليه. ونقول اصطلاحاً دمج مجموعتين من البيانات معاً أو ضمهما لنشير إلى أن كل مجموعة منها كانت تتمتع بوجود مستقل، ثم قمنا بوضعهما معاً لنكوِّن منهما تركيباً بيانياً جديداً له سمات الأصل ذاتها. ومعنى ذلك إذا كانت إحدى المجموعتين مرتبة وفق سياق معين (رقمياً أو هجائياً أو غير ذلك)، فلا بد أن تكون المجموعة الأخرى كذلك حتى تسهل عملية الدمج. وغالباً ما تستخدم هذه العملية بشكل خاص في الملفات

حيث يكون لدينا ملف قديم وملف للإضافات الجديدة (transactions) يتم وضعهما معاً في ملف واحد ليكونا بذلك ملفاً رئيساً جديداً.

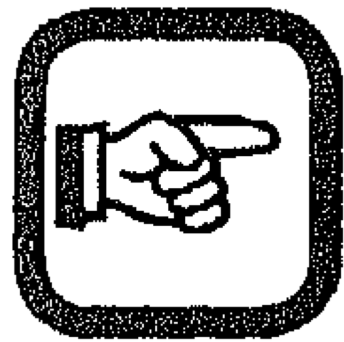


مثال (8)

بالإضافة إلى المصفوفة التي أشرنا إليها في الأمثلة السابقة، افترض أن المدرس لديه شعبة أخرى للمقرر نفسه ويريد دمج علامات الشعبتين معاً ليكون بذلك جدولاً واحداً لجميع طلبة الشعبتين. ولكي يتم ذلك لا بد من توفر مساحة كافية لسجلات طلبة الشعبتين، ومن هنا فإن المصفوفة الجديدة ينبغي أن تكون قادرة على استيعاب جميع الحالات. ونتيجة القيام بعملية الدمج هذه، سينتج لدينا مصفوفة مرتبة وفق الأصل.

هـ - الاضافة والحذف (Insertion and Deletion):

الإضافة في اللغة تعني الضم والإسناد. فنقول «انضاف الشيء إلى الشيء»: انضم أو أسند أو زيد إليه. والحذف هو عكس الإضافة، أي النقصان. فنقول «احتذف الشيء»: قطع بعضه أو أنقصه. وفي الاصطلاح نقول أضفنا قيمة بيانية جديدة إلى أحد التراكيب البيانية القائمة لنعني بذلك أن عدد القيم المضمنة في هذا التركيب البياني قد زاد. والعكس صحيح بالنسبة للحذف. ويستلزم القيام بعملية الإضافة أن يتوفر مكان شاغر في ذاكرة الحاسوب لكي نتمكن من تخزين القيمة الجديدة، وإلا وصلنا إلى وضع نطلق عليه اسم الفائض (overflow). أما بالنسبة لعملية الحذف فتستلزم وجود قيمة بيانية واحدة على الأقل، وإلا سنصل إلى وضع نطلق عليه اسم الخالي (underflow)، وهو عكس الفائض.

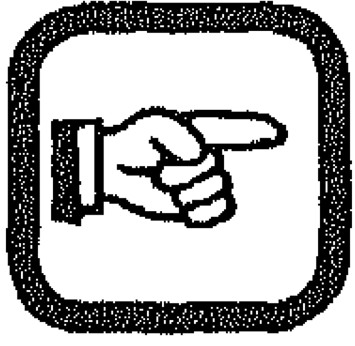


مثال (9)

باستخدام المصفوفة ذاتها التي أشرنا إليها في الأمثلة السابقة، دعنا نفترض أن أحد الطلبة سجل في المقرر المذكور متأخراً خلال مرحلة الانسحاب والإضافة، فمعنى ذلك أن اسمه لم يكن وارداً مع الطلبة الآخرين. وهذه الحالة تقتضي منا أن نقوم بإضافة سجل الطالب الجديد إلى المكان المناسب ضمن السجلات السابقة. وفي المقابل دعنا نفترض أن أحد الطلبة يرغب في الانسحاب من المقرر بعد أن سجل له. وهنا لا بد من إجراء عملية الحذف على سجل هذا الطالب. وقد تكون عملية الحذف في هذه الحالة منطقية كأن نضع علامة بجانب اسمه في حقل خاص بأن الطالب منسحب، وقد تتم عملية الحذف بطريقة مادية من خلال إعادة تنظيم السجلات.

و - التحديث (Updating):

حدَّث في اللغة نقيض قدم، والتحديث بهذا المعنى هو جعل الشيء مطابقاً لعصره. وفي الاصطلاح نقول قمنا بتحديث البيانات، بمعنى قمنا بإزالة صفة القدم عنها من خلال إجراء التغيرات المناسبة عليها. فبعض المعلومات تتقادم وتحلّ محلها معلومات أخرى حديثة، وعليه، لا بد أن نعكس ذلك في البيانات التي نقوم بتخزينها. فالتحديث بهذا المعنى هو إجراء التغييرات المناسبة على المعلومات المخزنة، ولذلك كثيراً ما يستعمل مصطلح التغيير (change) للتعبير عن هذه العملية.



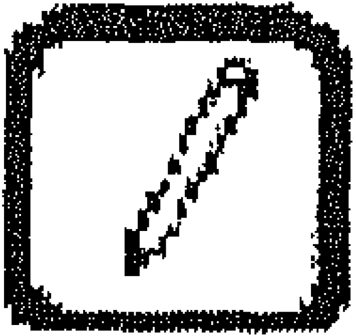
مثال (10)

لنفترض الآن أن الرقم الجامعي لأحد الطلبة قد تغير لأحد الأسباب. فمعنى ذلك أن الرقم الجامعي المخزن في مثال (9) السابق عن هذا الطالب لم يعد يعكس المعلومات الصحيحة. وعليه لا بد من إجراء عملية تحديث على الرقم القديم واستبداله بالرقم الجديد.

ز - عمليات أخرى:

بالإضافة إلى العمليات التي ذكرناها أعلاه، هناك بعض العمليات الأخرى التي تتم على بعض أنواع التراكيب البيانية وقد لا تتم على غيرها. ومثال ذلك تلك العمليات التي تجرى على سلاسل الرموز كالنسخ (copying)، والاستبدال (replacement)، والمطابقة النمطية (pattern matching)، وإيجاد الطول (length)، وغيرها.

وسيتم الحديث، عزيزي الدارس، عن هذه العمليات الخاصة وغيرها في أماكنها المناسبة خلال الوحدات القادمة.



تدريب (2)

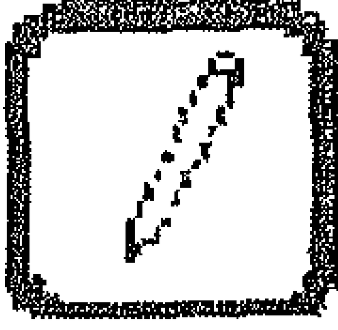
لديك التعبير الجبري التالي: $(7x + y)\ (5\ a - b)^3$ والمطلوب منك هو:

أ. اختيار أحد التراكيب البيانية لتمثيل هذا التعبير من بين التراكيب التي درستها.

ب. إعطاء التصور المنطقي لهذا التركيب البياني «باستخدام الرسم».

ج. توضيح طريقة التخزين في ذاكرة الحاسوب من خلال استخدام المصفوفات المتوازية «باستخدام الرسم».

تذكر أنك تعبر عن عملية الضرب باستخدام الرمز «*»، وأن الأقواس لا تدخل ضمن التمثيل المطلوب، وبعد أن تجيب على هذه النقاط، قارن بين إجاباتك بالإجابة المعطاة في نهاية الوحدة. فإذا وجدت فرقاً حاول أن تعيد دراسة الجزء الخاص بالتراكيب البيانية.



تدريب (3)

تأمل المصفوفتين التاليتين ثم أجب على الأسئلة المذكورة أدناه:

START = 5

FREE = 9

	INFO	LINK
1	Rabah	7
2	Qasim	8
3		0
4	Bilal	10
5	Hasan	2
6	Basim	1
7	Muna	0
8	Eman	4
9		3
10	Nour	6

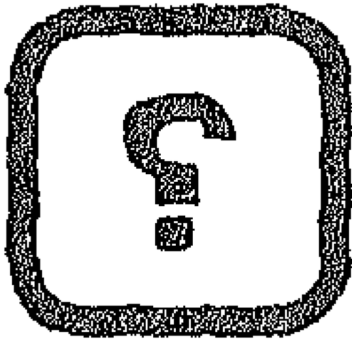
الشكل (15): طريقة تمثيل تركيب بياني في ذاكرة الحاسوب

أ. ما نوع التركيب البياني الذي يعبر عنه هذا الشكل؟

ب. أعط تصوراً منطقياً للتركيب البياني الممثل في الشكل أعلاه وذلك باستخدام الرسم.

ج. بين بالرسم العلاقة القائمة بين الأماكن الشاغرة (FREE) في الشكل أعلاه.

أسئلة التقويم الذاتي (3)



1. هناك ثلاثة أسباب تدعونا للاهتمام بالتراكيب البيانية، لخص هذه الأسباب باختصار من خلال ما درسته عن أهمية تركيب البيانات في هذه الوحدة.
2. اعتماداً على ما تكوّن الآن في ذهنك من معلومات عن التراكيب البيانية، ما أشهر هذه التراكيب؟

5. الخلاصة

الآن قد انتهيت، عزيزي الدارس، من دراسة الوحدة الأولى من هذا المقرر الدراسي، أتوقع أن تكون قادراً على التعامل مع المفاهيم التي تضمنتها هذه الوحدة. إن كان لديك أي شك في ذلك فإني أنصحك بالعودة إلى تلك الأجزاء التي تشعر أنك لم تستوعبها جيداً. وإذا كانت لديك أي تساؤلات، فالأفضل أن تتصل بمشرفك الأكاديمي لتوضيح النقاط الغامضة بالنسبة لك.

لقد حاولنا خلال هذه الوحدة أن نقدم إليك المفاهيم الأساسية المتصلة بتركيب البيانات، حاولنا أن نوضح هذه المفاهيم بالكثير من الأمثلة والصور التوضيحية والتدريبات التي تساعدك على استيعاب هذه المفاهيم. وقد ناقشنا الأنواع المختلفة للبيانات والتراكيب البيانية البسيطة والمركبة. كما وضحنا الفرق بين التركيب المنطقي والفيزيائي للبيانات.

خلاصة القول، فإن الأفكار التي عرضت في هذه الوحدة تهدف إلى تقديم موضوع تركيب البيانات بصفة عامة، وهناك الكثير من المفاهيم المتعلقة ببعض هذه الأفكار سيتم تفصيلها في الوحدات القادمة. وينبغي أن نتذكر، على أي حال، بأن موضوع تركيب البيانات هو من الموضوعات التي تمتزج فيها المسائل المنطقية والمادية، تتفاعل فيها نظرة المبرمج مع نظرة الجهاز بصورة متكافئة. ومهمة المبرمج الأساسية هي الوصول بالنهاية إلى اختيار تركيب بياني يستطيع أن يقوم بحل المشكلة على درجة مناسبة من الكفاءة، بالنسبة للمبرمج والجهاز والمشكلة المراد حلها.

6. لمحة عن الوحدة الدراسية الثانية

تتناول الوحدة الثانية من هذا المقرر موضوع تحليل الخوارزميات، ذلك بهدف المقارنة بينها من حيث وقت التنفيذ اللازم وذاكرة الحاسوب اللازمة. وهو موضوع وثيق الصلة بتراكيب البيانات، وذلك لأن إحدى خصائص تركيبة البيانات الجيدة هي استخدامها لخوارزميات كفؤة.

7. إجابات التدريبات

تدريب (1)

الشكل الوارد في التدريب يمثل التركيب المنطقي للقائمة الخطية X، أمّا التركيب الفيزيائي في لغة مثل لغة فورتران حيث لا تتوفر المؤشرات على النحو الذي نعرفه في لغة سي++ فيمكن أن يتخذ الصورة التالية:

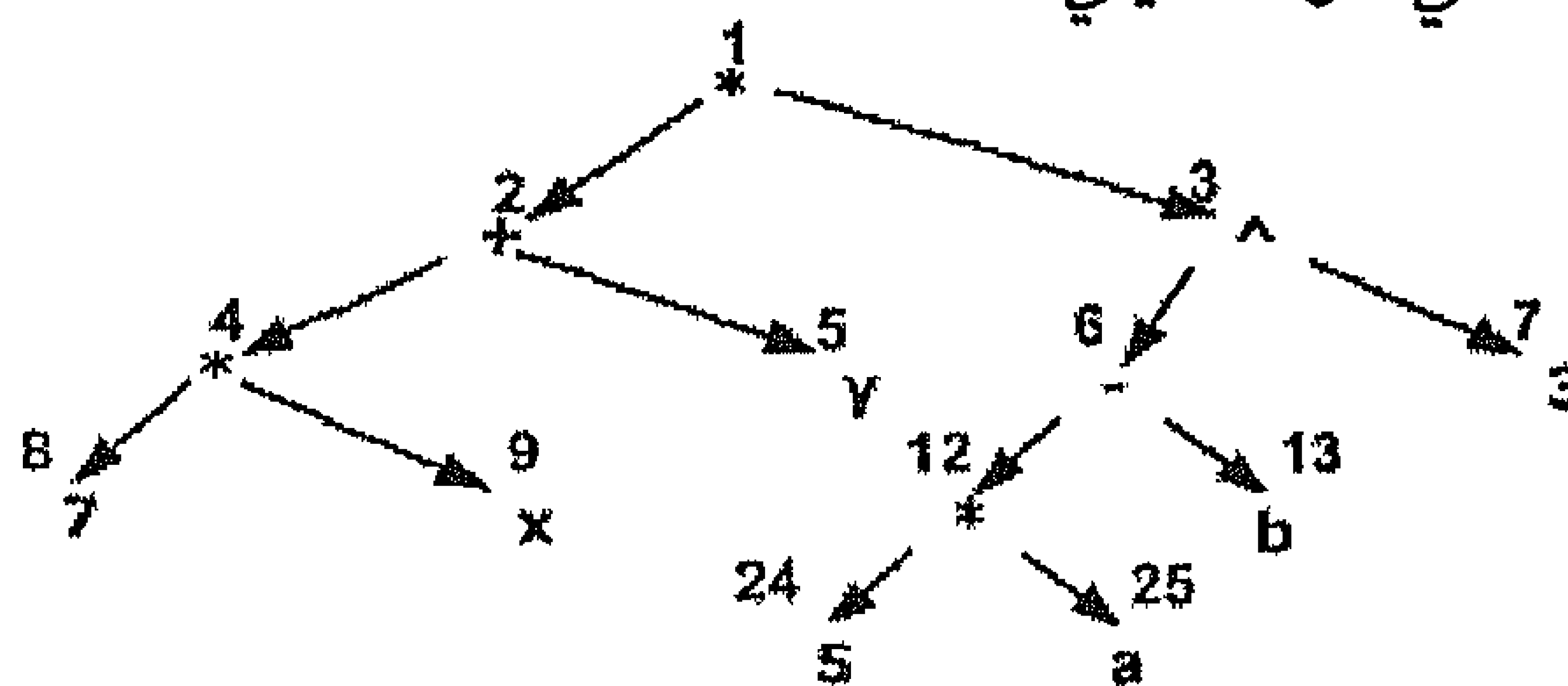
X = 5

	INFO	LINK
1	C	3
2		
3	D	0
4		
5	A	7
6		
7	B	1
8		

تدريب (2)

أ- التركيب البياني المناسب لذلك هو الشجيرة.

ب- التصور المنطقي هو كما يلي:



ج- طريقة التخزين باستخدام المصفوفات المتوازية هي كما يلي:

ROOT = 3

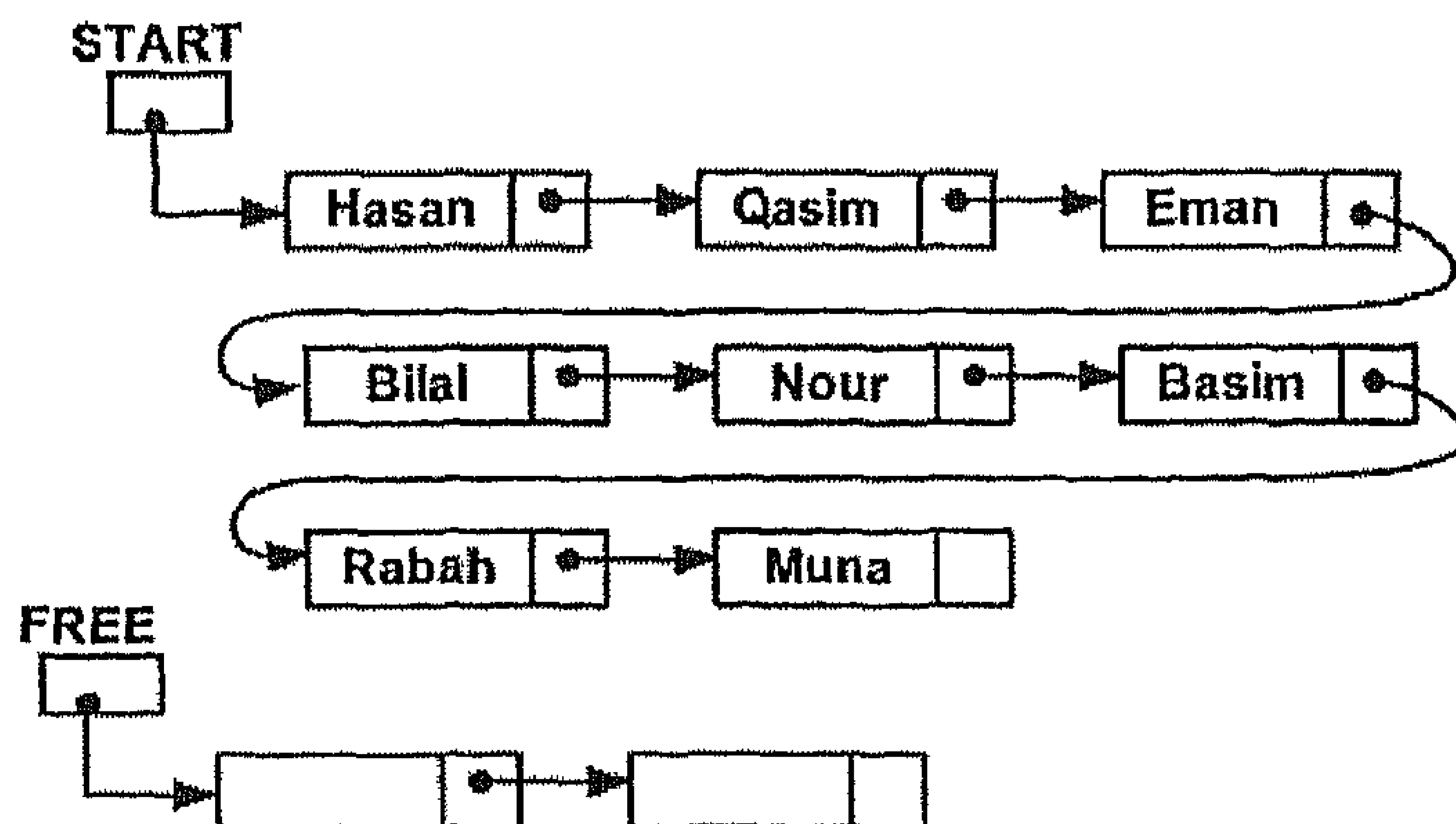
	INFO	LEFT	RIGHT
1	7	0	0
2	x	0	0
3	*	4	7
4	+	6	5
5	y	0	0
6	*	1	2
7	^	12	11
8	5	0	0
9	a	0	0
10	b	0	0
11	3	0	0
12	-	13	10
13	*	8	9

وكما تلاحظ، عزيزي الدارس، فإن لدينا ثلاث مصفوفات متوازية إحداها لتخزين عناصر الشجرة الواردة أعلاه، والأخريان لتخزين المؤشرات التي تصل العناصر ببعضها وتوضح علاقة التبعية بينها يميناً ويساراً.

تدريب (3)

أ- التركيب البياني الذي يعبر عنه هذا الشكل هو قائمة متصلة (linked list).

ب- التصور المنطقي لهذا التركيب البياني هو كما يلي:



وكما تلاحظ، عزيزي الدارس، فإن تخزين التركيب البياني الوارد في التدريب قد انطوى على مصفوفتين متوازيتين: إحداهما لتخزين القيم INFO والأخرى لربط العناصر بعضها ببعض. وكما هو واضح. فإن العنصر الأوّل هو «Hasan». وقد علمنا ذلك من خلال المؤشر START وهو مخزن في الموضع رقم «5»، ومن خلال الموضع المقابل في المصفوفة LINK نستطيع أن نعرف العنصر التالي في التسلسل وهو المخزن في الموضع رقم «2»، أي «Qasim».. وهكذا.

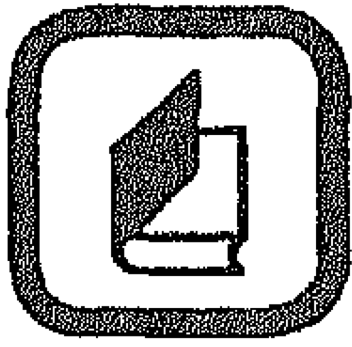
8. مسرد المصطلحات

- **أنماط بيانية بسيطة Simple data types**: هي البيانات التي تنطوي على قيمة بيانية واحدة في أدنى مستويات التعبير عن الأشياء والأحداث المحيطة بنا. وهي تضم القيم العددية الصحيحة والحقيقية والرموز على اختلاف أشكالها والقيم المنطقية وما شابه ذلك.

- **أنماط بيانية مركبة Structured data types**: هي التراكيب البيانية التي ينطوي بناؤها وتعريفها على أكثر من قيمة بيانية، ولها صورة تنظيمية معينة سواء من الناحية المنطقية أو الناحية الفيزيائية في ذاكرة الحاسوب. ومثال ذلك المصفوفات، والسجلات، والمجموعات، وسلاسل الرموز وغير ذلك مما يتوفر في لغات الحاسوب أو مما يوفره المبرمج نفسه.

- **تراكيب البيانات Data structures**: هو الموضوع الذي يتناول الطرق والأساليب المختلفة التي يمكن من خلالها ترجمة التصور المنطقي للبيانات كما يراها المبرمج إلى تمثيل مناسب في ذاكرة الحاسوب.

- **التمثيل المتصل Linked representation**: هو تخزين العناصر المختلفة لتركيب بياني معين في أماكن غير متجاورة في الذاكرة واستخدام مؤشرات الربط للوصل بين العناصر والإشارة إلى عناوينها. فكل عنصر يشير إلى ما يليه على ضوء التصور المنطقي الخاص بالتركيب البياني المعني. وقد يتم هذا التمثيل باستخدام المصفوفات المتوازية.



9. المراجع

1. Aho, Alfred; Hopcroft, John E.; and Ulman, Jeffrey D., Data Structures and Algorithms. Reading (USA): Addison-Wesley, 1983.
2. Beidler, John, An Introduction to Data Structures. Boston: Allyn And Bacon, 1982.
3. Lewis, T.F., and Smith, M.Z., Applying Data Structures. Atlanta (USA): Houghton Ifflin, 1976.
4. Lipschutz, Seymour, Schaum's Outline of Theory and Problems of Data Structures. New York: McGraw-Hill, 1986.
5. Tremblay, J.P, and Sorenson, P.G., An Introduction to Data Structures with Applications. New York: McGraw-Hill, 1976.

الوحدة
الثانية

مبادئ تحليل الخوارزميات

محتويات الوحدة

1 : المقدمة

1.1 تمهيد

أهلاً بك، عزيزي الدارس، في الوحدة الثانية من كتاب «تركيب البيانات وتصميم الخوارزميات» وهي بعنوان «مبادىء تحليل الخوارزميات»، وفي هذه الوحدة سنتعرف على المبادئ المتبعة لتحليل الخوارزميات، وما نقصده عند قولنا تحليل الخوارزمية هو دراستها والتعرف على مدى كفاءتها من حيث وقت التنفيذ اللازم وذاكرة الحاسوب اللازمة. وهو موضوع وثيق الصلة بتركيب البيانات، إذ يحتاج المرء لاختيار تركيبة البيانات المناسبة إلى الإلمام بكفاءة أو نجاعة الخوارزميات الضرورية لمعالجتها، وعليه فإن موضوعا تركيب البيانات وتحليل الخوارزميات موضوعان مهمان ومتلازمان. وقد حاولنا في هذه الوحدة توضيح المفاهيم الرئيسة مستخدمين العديد من الأمثلة والتدريبات المحلولة في نهاية الوحدة، آملين أن تستفيد وتستمتع بدراسة هذه الوحدة من الكتاب.

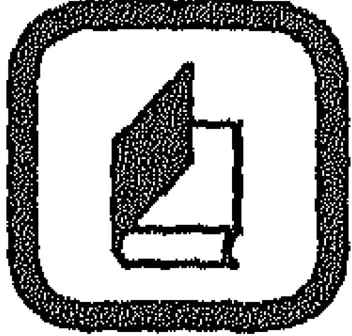
2.1 أهداف الوحدة

ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1. توضح مبادئ تحليل الخوارزميات.
2. تقارن بين الخوارزميات المختلفة، وتختار الأفضل منها من حيث مقدار وقت التنفيذ اللازم، ومساحة الذاكرة اللازمة في أثناء التنفيذ.
3. تفرز البيانات (ترتبها ترتيباً تصاعدياً) مستخدماً بعض خوارزميات الفرز.

3.1 أقسام الوحدة

تتكون هذه الوحدة من أربعة أقسام رئيسة ترتبط بقائمة الأهداف السابقة. نناقش في القسم الأول منها أهمية تحليل الخوارزميات ونتعرف في القسم الثاني على طريقة رياضية لتحليل البيانات، وفي القسم الثالث نناقش العديد من مقاييس تنفيذ مميزة ومتكررة، وهذه الأقسام مرتبطة بتحقيق الهدفين الأول والثاني. والقسم الأخير يناقش ثلاث خوارزميات خاصة بعملية الفرز (أي ترتيب البيانات) وهذا القسم مرتبط بالهدفين الثاني والثالث.



4.1 القراءات المساعدة

عزيزي الدارس، حاول الانتفاع بالقراءات الآتية لاتصالها القوي والمباشر بموضوع الوحدة، ولا شك أن انتفاعك بها سيجعلك أكثر قدرة على استيعاب ما فيها من معلومات، كما سيعمق فيك القدرة على التصور والتحليل والاستنتاج:

1. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice -Hall. 2000.
2. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison Wesley, 1999.

2. أهمية تحليل الخوارزميات

تذكر، عزيزي الدارس، أن الخوارزمية هي الخطوات اللازمة لحل مسألة ما، وقد تكتب هذه الخوارزمية باللغة العربية أو الإنجليزية، أو يعبر عنها بالرسم وسنستخدم لغة شبيهة بلغة سي لكتابة الخوارزميات في هذا المقرر.

من المهم كي تتمكن من اختيار تركيب البيانات المناسبة أن تكون قادراً على تحليل الخوارزمية الخاصة بهذا التركيب ومقارنة هذه الخوارزميات لاختيار الأفضل أو الأنجع، إذ أن اختيارك لخوارزمية معينة يعني اختيارك بالضرورة لمجموعة من الخوارزميات اللازمة لمعالجتها، والتعامل معها.

وعندما تقارن بين الخوارزميات فإن المقاييس التي تستخدمها في المقارنة هي:

1. مقدار وقت الحاسوب اللازم لتنفيذها.
2. مساحة ذاكرة الحاسوب التي تحتاج إليها الخوارزمية.
3. وضوح الخوارزمية وبساطتها.

للأسف، عزيزي الدارس، فقد تتعارض هذه المقاييس إذ قد تكون الخوارزمية الأبسط هي الأقل نجاعة من حيث الوقت والذاكرة اللازمين لتنفيذ الخوارزمية، وبالمثل قد تكون الخوارزمية التي تتطلب وقتاً قليلاً لتنفيذها هي الخوارزمية التي تستخدم ذاكرة كبيرة. وعليه فإننا في كثير من الأحيان قد نلجأ إلى الحلول الوسط آخذين بعين الاعتبار طبيعة الظروف التي ستنفذ فيها الخوارزمية. على سبيل المثال، هل ستنفذ في وجود عميل ينتظر النتيجة، وعليه يكون وقت التنفيذ هو العامل الأهم.

إن تحويل الخوارزمية إلى برنامج وتنفيذ هذا البرنامج لحساب الوقت الحقيقي (بالدقائق والثواني) اللازم لتنفيذ خوارزمية معينة لا تعطي دائماً فكرة جيدة عن جودة الخوارزمية، وذلك لكون وقت التنفيذ الحقيقي يعتمد على عوامل مختلفة ليست لها علاقة بجودة الخوارزمية. هذه العوامل هي:

1. سرعة الحاسوب إذ أن الوقت الحقيقي يعتمد على سرعة الحاسوب المستخدم لتنفيذ البرنامج.
2. كمية البيانات، إذ أنه من الطبيعي أن يعتمد الوقت الحقيقي على كمية البيانات المراد معالجتها، فكلما ازدادت كمية البيانات ازداد وقت التنفيذ.
3. عوامل أخرى لها علاقة بطريقة تنفيذ الخوارزمية (تحويلها إلى برنامج في لغة برمجة معينة) كلغة البرمجة المختارة والمبرمج الذي كتب البرنامج ومقدار خبرته.

4. لهذه الأسباب مجتمعة، كان لا بد من تطوير طريقة رياضية أكثر تجرداً لتحليل وقت التنفيذ.



أسئلة التقويم الذاتي (1)

1. ما المقاييس التي تستخدم في المقارنة بين الخوارزميات؟
2. وضح العوامل التي تجعل قياس الوقت الحقيقي لتنفيذ البرامج غير مناسب دائماً لمقارنة الخوارزميات.

3. تحليل الخوارزميات رياضيا

لاحظ، عزيزي الدارس، أن وقت تنفيذ الخوارزمية يعتمد على عدد خطوات الخوارزمية، لذلك فإن أول خطوة في تحليل الخوارزمية هي التعبير عن عدد الخطوات في الخوارزمية بوساطة اقتران بدلالة حجم البيانات (n).

لندرس على سبيل المثال الخوارزمية التالية التي تطبع عناصر مصفوفة وتجد مجموع هذه العناصر:

```
int sum=0;
for ( int i=0;i<n;i++)
  {  sum +=A[i];
      Cout<<A[i]; }
```

لاحظ، عزيزي الدارس، أنك تستطيع التعبير عن عدد خطوات هذه الخوارزمية بدلالة اقتران رياضي بدلالة n. لاحظ وجود خطوة واحدة قبل الدوران تنفذ مرة واحدة وخطوتان داخل الدوران تنفذان n من المرات لذا فإن عدد الخطوات اللازمة T(n) لتنفيذ الخوارزمية هي:

$$T(n) = 1 + 2n$$

لاحظ كذلك أننا نتعامل مع الخطوات وكأن كلاً منها تحتاج إلى نفس المقدار من الوقت للتنفيذ وذلك لتسهيل مهمتنا؛ بالرغم من أن ذلك قد لا يكون صحيحاً فعملية الضرب قد تأخذ وقتاً أكبر من عملية الجمع، على سبيل المثال.

وللتعبير الدقيق عن عدد خطوات خوارزمية معينة قد نحصل على اقتران طويل ومعقد ليست كل عوامله مهمة عندما تكون كمية البيانات كبيرة، أو بكلمات أخرى عندما تكون قيمة n كبيرة. في مثالنا أعلاه واضح أن الواحد عامل يمكن تجاهله إذ أن تنفيذ خطوة بسيطة واحدة في الحاسوب يأخذ وقتاً قصيراً يمكن تجاهله خاصة، ونحن نهتم بمدى نجاعة الخوارزمية لكميات كبيرة من البيانات. ولتبسيط الاقتران أعلاه نذهب أبعد من ذلك ونتجاهل الثوابت حتى ولو كانت مستخدمة في عمليات ضرب كالرقم 2 في مثالنا أعلاه.

وقد لجأ محللو الخوارزميات إلى طريقة رمزية (notation) تدعى طريقة ترميز O الكبيرة (big-O notation) لاستخلاص العوامل المهمة في التعبير عن كفاءة الخوارزمية. لنفترض أن هنالك اقتراناً F(n) معرفاً على الأرقام غير السالبة بحيث يكون:

$$T(n) <= c * F(n)$$

عندما تكون: $n >= m$

حيث m وc هما ثابتان عدديان. عندها نستطيع القول بأن الخوارزمية السابقة هي ((O(F(n .

أي أنه إذا كان عدد خطوات الخوارزمية ((T(n) أقل أو يساوي حاصل ضرب الثابت c في f(n) عندما تكونn قيمة كبيرة أكبر من الثابت m عندها نستطيع القول بأن الخوارزمية هي ((O(F(n ويدعى الثابت c بثابت التناسب Constant of proportionality.

لنعود إلى الخوارزمية السابقة ونحاول إيجاد F(n) المطلوب، لاحظ، عزيزي الدارس، أن:

$$T(n) = 1 + 2n <= 2n$$

عندما تكون: $n > 2$

أي أن قيمة c (ثابت التناسب) هو 2 وأن قيمة m هي 2 وf(n) هي n وعليه نستطيع القول أن الخوارزمية هي O(n). وهي طريقة مختصرة للقول بأن وقت التنفيذ اللازم للخوارزمية ينمو بعلاقة خطية نسبة إلى حجم البيانات.

والآن لندرس كيفية تحليل خوارزمية (أخرى باستخدام طريقة الترميز O الكبيرة)

```
int s=0;
for  ( int k=0;k<n;k++)
 {
 for ( int j=0;j<n;j++)
   {cout<<k*j;
      s +=k*j;
```

لاحظ أن T)n) لهذه الخوارزمية هو

$$T(n) = 2n^2 + 1$$

وذلك لوجود دورانين أحدهما داخل الآخر. وأن:

$$22n^2 + 1 <= 3n^2$$

عندما تكون: $n >= 2$

وعليه فإن الخوارزمية تكون $O(n^2)$. أي أن وقت التنفيذ له علاقة تربيعية بحجم البيانات.

إن طريقة الترميز O الكبيرة تتجاهل الثوابت، وبالتالي هي لا تفرق بين خوارزميات

عدد خطواتها n^2 و$2n^3$ أو حتى 100n2 حيث تعتبرهم جميعا $O(n^2)$، وعليه فهي قد لا تكون طريقة دقيقة للمقارنة بين خوارزميات من الدرجة نفسها، ولكنها بالتأكيد طريقة جيدة للمقارنة بين خوارزميات من درجات مختلفة، على سبيل المثال $O(n)$ أفضل كثيراً من $O(n^2)$ مهما كانت الثوابت التي تجاهلناها حينما تكون قيمة n كبيرة بدرجة كافية.

لاحظ أيضاً، عزيزي الدارس، أننا أصلاً لم نحسب الثوابت بدقة حينما حسبنا $T(n)$ فلقد اعتبرنا أن كلاً من خطوات الخوارزمية تحتاج إلى الوقت التنفيذي نفسه.

وهذا ليس دائماً صحيحاً، كذلك أن هنالك العديد من الخطوات التي تجاهلناها، فجملة الدوران (for) على سبيل المثال هي جملة مركبة تحتوي على العديد من الخطوات كالمقارنة مع القيمة النهائية وزيادة العداد ومع ذلك تجاهلناها تماماً. ولعدم اهتمامنا بالثوابت فإننا نركز على الخطوة أو العملية الأكثر تكراراً في الخوارزمية عند تحديد الاقتران $T(n)$. وتسمى هذه العملية بالعملية الحرجة (The Critical Operation) ونتجاهل بقية العمليات (أو الخطوات).

وغالباً ما يعتمد عدد تنفيذ خطوات خوارزمية معينة على طبيعة البيانات، لذلك عادة ما نحلل الخوارزمية آخذين ثلاثة سيناريوهات بعين الاعتبار: الحالة الأفضل (Best case) وهي الحالة التي تحتاج إلى تنفيذ أقل عدد من الخطوات، الحالة الأسوأ (Worst case) وهي الحالة التي تحتاج فيها الخوارزمية إلى تنفيذ أكبر عدد من الخطوات، والحالة الوسطى (Average case) وهي الحالة التي تتطلب تنفيذ عدد معتدل من الخطوات. على سبيل المثال، خوارزمية البحث الخطي التالية التي تبحث عن قيمة معينة X في مصفوفة A مكونة من n من العناصر

```
I=0;
Found=false;
while  (I<n  &&  !found)
 {if  (A[I]==X)  found=true;
   else I++;
```

قد تنفذ عملية مقارنة واحدة في الحالة الأفضل، وذلك إذا كانت القيمة التي نبحث عنها مساوية للقيمة المخزنة في العنصر الأول وفي هذه الحالة تكون الخوارزمية $O(1)$. وقد تحتاج الخوارزمية إلى n من عمليات المقارنة إذا كانت القيمة التي نبحث عنها هي العنصر

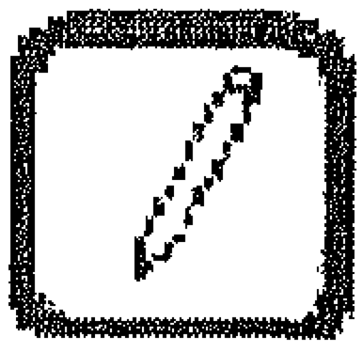
الأخير في المصفوفة أو لم تكن موجودة في المصفوفة، وفي هذه الحالة تكون الخوارزمية .O(n)

والآن ماذا عن الحالة الوسطى؟ إن تحليل الحالة الوسطى هي العملية الأعقد، وذلك لكونها تعتمد على معلومات إحصائية عن احتمال وجود القيمة في الموقع الأول في المصفوفة أو في الموقع الثاني أو الثالثة أو الأخير.

وإذا افترضنا توزيعاً متكافئاً (Uniform distribution) للبيانات فإن احتمال وجود القيمة المبتغاة هو (p=1/n) وبالتالي فإن متوسط عدد عمليات المقارنة وبالتالي زمن تنفيذ الخوارزمية يحسب بالعلاقة التالية:

$$C(n)=1(1/n)+2(1/n)+...+n(1/n)$$
$$= (n+1)/2$$

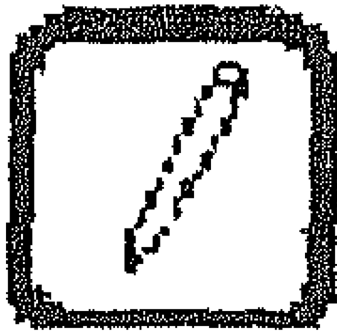
وعليه تكون الحالة الوسطى أيضاً O(n) وغالباً ما نركز جلّ اهتمامنا على تحليل الحالة الأفضل والحالة الأسوأ، ونتجاهل الحالة الوسطى لصعوبة تحليلها بطريقة دقيقة، ومن الآن فصاعداً، في هذا المقرر، سنتجاهل تحليل الحالة الوسطى.



تدريب (1)

استخدم طريقة ترميز O الكبيرة لتحليل الخوارزمية التالية:

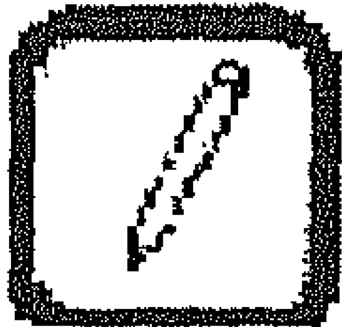
```
for (int k=1; k<=N*N; k++)
  {cout<<k<<endl;
  for (int j=1; j<=N/2; j++)
   {m=k*j;
   cout<<setw(4)<<k
     <<setw(4)<<j
     <<setw(4)<<m<<endl;}
```



تدريب (2)

استخدم طريقة ترميز O الكبيرة لتحليل الخوارزمية التالية:

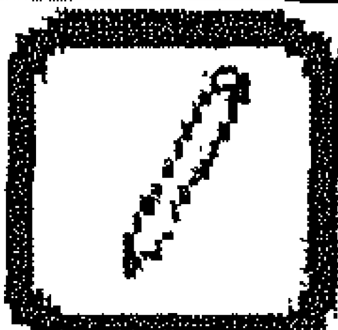
```
for  (int i=1; i<=300; i++)
    {cout<<i*i<<endl;
```



تدريب (3)

استخدم طريقة ترميز O الكبيرة لتحليل الخوارزمية التالية:

```
k=1;
 do {j=1;
      do
        {cout<<j<<»\n»;
         j=2*j; }
   while(j<=N);
   k++;
 }
 while (k<=N);
```



تدريب (4)

استخدم طريقة ترميز O الكبيرة لتحليل الخوارزمية التالية:

```
for (int i=1; i<=N; i++)
 cout<<i<<endl;
 for (int j=1; j<=N; j++)
  cout<<j<<endl;
```

4. مقاييس تنفيذ مميزة ومتكررة

على الرغم من أنه ليست هناك قواعد بسيطة ومحددة يمكن أن نسير بموجبها في تحديد زمن تنفيذ البرنامج وقياسه. فإن هناك بعض المقاييس الزمنية المميزة التي تظهر بشكل متكرر. وأكثر هذه المقاييس المميزة شيوعاً هي:

O (1) <O (log n) < O(n) <O (n log n)

<O (n2) < O (n3) <O (2n)

وكما تلاحظ فقد رتبنا هذه المقاييس حسب درجة قوتها من اليسار إلى اليمين. فلكل منها ميزته على الآخر، ويظهر الفرق بينها جلياً كلما زاد حجم n كما يتضح من الجدول (1) أدناه. وفيما يلي سنحاول، عزيزي الدارس، التعليق على هذه المقاييس.

جدول (1): معدل النمو في العلاقة بازدياد حجم n

n	log n	n	n log n	n^2	n^3	2^n
1	0	1	1	1	1	2
2	1	2	2	4	8	4
4	2	4	8	16	64	16
8	3	8	16	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,786	Billions
64	6	64	384	4,096	262,144	∞

أ. الزمن الثابت: (constant computing time)

زمن الحوسبة الثابت يشار إليه بالعلاقة O(1)، أي أن الوقت المطلوب للعملية لا يتغير. فكما أشرنا من قبل، تستغرق جملة الإسناد وقتاً ثابتاً، وكذلك جملة الكتابة أو القراءة، وغير ذلك كعمليات الوصول إلى أحد عناصر المصفوفة، أو استبدال قيمة بقيمة أخرى في عمليات الفرز، فكل هذه العمليات تمتاز بوقت ثابت.

ب. الزمن الخطي: (linear time)

إن الخوارزمية التي يشار إليها بالعلاقة O(n) هي خوارزمية ذات زمن يتناسب طردياً مع حجم المشكلة، أي مع أي زيادة تحدث على حجم البيانات. فعلى سبيل المثال، إذا

أردنا أن نجمع القيم المخزنة في عناصر إحدى المصفوفات فإن إيجاد أحد العناصر في قائمة متصلة (linked list) سيأخذ وقتاً خطياً، لأن من المحتمل أن نزور جميع عناصر القائمة.

ج. الزمن اللوغاريتمي: (logarithmic time)

إن الخوارزمية ذات الكفاءة اللوغاريتمية، والتي نشير إليها بالعلاقة O(log n) تقوم بعمل أكثر من الخوارزمية ذات الوقت الثابت، وتحتاج بالتالي وقتاً أطول منها ولكنه أقل من الوقت الذي تستغرقه خوارزمية ذات وقت خطي. فالوصول إلى أحد عناصر إحدى الشجيرات الثنائية الاستقصائية (binary search tree) على سبيل المثال، قد يستغرق وقتاً لوغاريتمياً. وهناك أيضاً بعض التراكيب الدورانية التي يعبر عنها بوقت لوغاريتمي، ومن أمثلة ذلك الجملتان الدورانيتان التاليتان. فالجملة الأولى تستغرق O(log n). أما الثانية فتستغرق زمناً مقارباً للعلاقة (log n).

```
(a)  k=1;
     while (k<=n)
        k +=k/2;
(b)  k=n;
while (k!=0)
   k =k/2;
```

وهناك بعض الحالات التي تستنفذ زمناً خطياً لوغاريتمياً. وفي هذه الحالات تصبح العلاقة O(n log n). ويزيد معدل النمو بشكل يفوق الزمنين الخطي واللوغاريتمي كما هو واضح في الجدول (1). ومن بين الحالات التي ينطبق عليها هذا الوضع ترتيب مجموعتين من العناصر في مجموعة واحدة فيما يعرف باسم الفرز الدمجي أو المزجي (merge sort) التركيب الدوراني التالي مثال واضح على هذا الزمن الخاص.

```
for (i=1; i<=n; i++)
   j=i;
  while (j!=0)
      j=j/2;
```

د. الزمن المتعدد الحدود: (polynomial time)

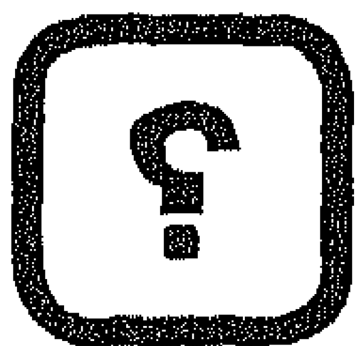
إذا كانت العلاقة الزمنية محكومة بمتغير متعدد الحدود، فإن نمو العلاقة يزداد بمعدل تربيعي (quadratic) أو تكعيبي (cubic)، كما هو واضح في الجدول(1) O(n2)

و$O(n^3)$ أو$O(nc^2)$. فالخوارزمية ذات الكفاءة التربيعية تقوم بأعمال أكثر مما تقوم به خوارزمية ذات وقت خطي، ومن هنا تستنفذ وقتاً أكثر. وبالمثل فإن الخوارزمية ذات الوقت التكعيبي تستغرق وقتاً أكثر من الخوارزمية ذات الزمن التربيعي. فإذا قمنا بتخزين قيمة ابتدائية في كل عنصر من عناصر مصفوفة ذات ثلاثة أبعاد، على سبيل المثال، فإن ذلك سيستغرق زمناً تكعيبياً (O (n3.

هـ . الزمن الأسي: (exponential time)

إن الخوارزمية التي تستنفذ مثل هذا الزمن يشار إليها بالعلاقة $O(a^n)$، وكلما زادت قيمة n ظهر فرق كبير بين الزمن الأسي والزمن الخاص بالحدود المتعددة كما هو واضح في الجدول السابق. وبصفة عامة، فإن الخوارزميات ذات الزمن الأسي (exponential) غير عملية، ولا يُنصح باستخدامها إلا في حالات خاصة عندما يكون حجم n صغيراً جداً، ذلك لأن كلفتها كبيرة ويصعب الوفاء بها. وإذا استطعت أن تجد خوارزمية متعددة الحدود بدلاً من خوارزمية أخرى ذات زمن أسي فسيكون ذلك انجازاً كبيراً. وكقاعدة عامة، ينبغي عدم اختيار خوارزمية يزيد معدل كفاءتها الزمنية عن$O(n^3)$. والتركيب الدوراني التالي هو أحد الأمثلة التي نصل فيها إلى الزمن الأسي، حيث يتم تنفيذ جملة الدوران الداخلية بمعدلI(I) .

```
for ( int i=1; i<=n; i++ )
 { aingr=1.0/(i*i);
   x=0;
 while (x<=i)
     x +=aingr;        }
```



أسئلة التقويم الذاتي (2)

عدد أكثر المقاييس شيوعاً في تحديد زمن تنفيذ البرامج.

5. خوارزميات الفرز

هنالك، عزيزي الدارس، العديد من الخوارزميات لفرز قوائم (lists) من البيانات أي لترتيبها ترتيباً تصاعدياً أو تنازلياً. وسندرس في هذا القسم ثلاث خوارزميات للفرز، ونقارن بينها ليس فقط لتعريفك على بعض خوارزميات الفرز، بل أيضا لتطبيق ما تعلمناه من طرق تحليل الخوارزميات للمقارنة بينها.

1.5 خوارزمية الفرز الفقاعي The Bubble Sort Algorithm

```
for ( int  i=0;  i<n-1;  i++ )
     for ( int j=i+1; j>0; j - - )
        if (a[j-1]<a[j] )
{temp=a[j-1];
 a[j-1]=a[j];
 a[j]=temp;
  }
```

هذه خوارزمية بسيطة تقوم بمقارنة كل عنصرين متجاورين في المصفوفة، واستبدالهما إذا وجد أن العنصر الأول أكبر من الذي يليه. لاحظ أن الخطوة الحرجة (الأكثر تكراراً) في هذه الخوارزمية هي عملية المقارنة (جملة الـ if)، وعدد تكرارها هو المجموع.

$$T(n) = (n - 1) + (n - 2) + (n - 3) + ... + 3 + 2 + 1 = \frac{(n - 1)\,(n)}{2}$$

وبهذا يتضح أن الخوارزمية هي $O(n^2)$

2.5 خوارزمية الفرز الانتقائي The Selection Sort Algorithm

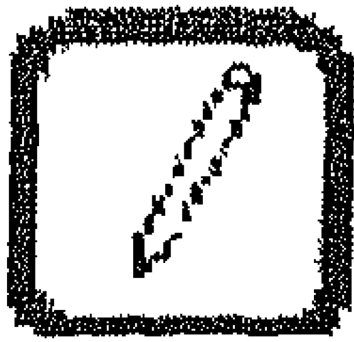
```
for (int i=0; i<n-1; i++)
    {//find smallest element in the unsorted part
     min=x[i];
     position=i;
     for (int j=i+1; j<=n; j++)
      if (x[j]<min)
{min=x[j];
 position=j;
 }
//Swap the smallest element (x[position] with x[i]
```

```
temp=x[i];
x[i]=x[position];
x[position]=temp;
  }
```

تقوم الخوارزمية بالفرز الانتقائي بترتيب عناصر المصفوفة بتكرار عمليتين رئيستين هما:

1. إيجاد أقل عنصر في الجزء غير المرتب من المصفوفة.
2. استبدال ذلك العنصر مع أول عنصر في الجزء غير المرتب.

في المرحلة الأولى نجد أصغر عنصر في كامل المصفوفة ثم نستبدله مع أول عنصر فيتبقى نتيجة لذلك n-1 من العناصر غير المرتبة وهي العناصر ذات الترتيب ... n, 3, 2 ثم تعاد الكرّة وتجد الخوارزمية أقل عنصر في هذه العناصر، ثم تستبدله بالعنصر الثاني وهكذا تبقى العناصر 3 ,4 ,.... ,n غير مرتبة. عند إعادة تنفيذ هاتين العمليتين n - 1 ينتج عن ذلك ترتيب جميع عناصر المصفوفة.



تدريب (5)

لماذا يكفي تكرار الدوران الخارجي n-1 من المرات فقط في خوارزمية الفرز الانتقائي؟

لاحظ، عزيزي الدارس، أن العملية الحرجة في هذه الخوارزمية هي أيضاً عملية المقارنة وهي أيضاً موجودة داخل دورانين متداخلين (nested loops)، لاحظ أيضاً أن عدد تكرار جملة المقارنة هو المجموع:

$$T(N) = (N - 1) + (N - 2) + (N - 3) + (N - 4) + ... + 3 + 2 + 1 = \frac{N^*(N-1)}{2}$$
$$= \frac{1}{2}N^2 - \frac{1}{2}N$$

وعليه فإن خوارزمية الفرز الانتقائي هي أيضاً $O(n^2)$.

ولكن هل يعني ذلك أن الخوارزميتين بنفس الجودة؟ للإجابة على هذا السؤال لاحظ أن خوارزمية الفرز الفقاعي تكرر عملية التبديل (Swaps) بعدد أكبر من المرات بالمقارنة مع خوارزمية الفرز الانتقائي. صحيح أننا اعتبرنا عملية المقارنة العملية الحرجة ولم نأخذ بعين الاعتبار عملية التبديل الأقل تكراراً. إلا أن عملية التبديل قد تحتاج إلى وقت تنفيذ أكبر بكثير من عملية المقارنة خصوصاً إذا كان كل عنصر في المصفوفة هو سجل (record)

كبير يحتوي على العديد من الحقول الرمزية على سبيل المثال. وبناءً عليه وللمقارنة الدقيقة بين الخوارزميتين علينا أخذ عدد تكرار عملية التبديل بعين الاعتبار. قارن، عزيزي الدارس، موقع عملية التبديل في خوارزمية الفرز الفقاعي وخوارزمية الفرز الانتقائي. لا شك أنك لاحظت وجود عملية التبديل في خوارزمية الفرز الانتقائي موجودة داخل دوران واحد يتكرر n-1 من المرات. إذاً نستطيع القول بأن هذه الخوارزمية هي O(n) بالنسبة لعدد عمليات التبديل التي تجريها.

ولكن الأمر أعقد قليلاً بالنسبة لعملية التكرار في خوارزمية الفرز الفقاعي إذ أن عملية التبديل موجودة داخل دورانين متداخلين. وفي أسوأ الأحوال فإن عدد عمليات التبديل يكون مساوياً لعدد عمليات المقارنة أي:

$$(n-1)+(n-2)+(n-3)+\ldots+3+2=\frac{1}{2}n^2-\frac{1}{2}n-1$$

وبهذا تكون خوارزمية الفرز الفقاعي O(n2) بالنظر إلى عدد التبادل وذلك في أسوأ الأحوال، أي عندما تكون المصفوفة مرتبة ترتيباً عكسياً. إذا تكون في هذه الحالة خوارزمية الفرز الفقاعي أسوأ بكثير من خوارزمية الفرز الانتقائي من حيث عدد عمليات التبديل الواجب إجراؤها. وتستطيع، عزيزي الدارس، ملاحظة أن خوارزمية الفرز الفقاعي في المقابل لن تجري أي عملية تبديل إذا كانت المصفوفة مرتبة أصلاً. بينما ستجري خوارزمية الفرز الانتقائي N من عمليات التبديل (لاحظ أن كل عنصر سيستبدل بنفسه في هذه الحالة).

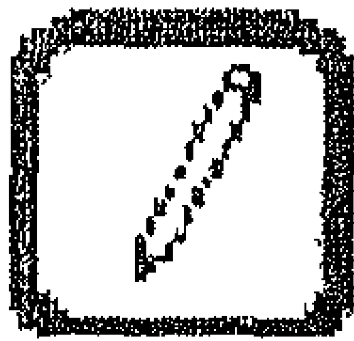
3.5 خوارزمية الفرز الإدخالي The Insertion Sort Algorithm

```
for (int k=1; k<n; k++)
    {i=k;
     inserted=false;
     while (i>0 && !inserted)
         {if (a[i]<a[i-1])
            {//swap a[i] with a[i-1]
               temp=a[i];
    a[i]=a[i-1];
    a[i-1]=temp;
    }
   else inserted=true;
    i--;
  }
  }
```

إن من عيوب خوارزميتي الفرز الانتقائي والفرز الفقاعي أن كلاً منها سوف تجري نفس العدد من المقارنات بغض النظر عن الحالة الابتدائية للمصفوفة حتى لو كانت المصفوفة مرتبة أصلاً. أما خوارزمية الفرز الإدخالي فإنها تستطيع ملاحظة فيما إذا كانت خوارزمية مرتبة أو شبه مرتبة أصلاً وتنفذ في تلك الحالة عدداً أقل من عمليات المقارنة.

إن الخوارزمية تقوم بترتيب العناصر (ابتداء من العنصر الثاني) بإدخال كل عنصر في المكان الصحيح نسبة إلى جميع العناصر التي سبق ترتيبها (إدخالها). والمكان الصحيح لأي عنصر هو بالطبع المكان الذي تكون فيه جميع القيم التي تسبقه أقل منه وجميع القيم التي تليه أكبر منه.

لاحظ، عزيزي الدارس، أن الخوارزمية أعلاه تدخل العنصر الثاني في المكان الصحيح في أول دوران أي عند (k=2). أي يصبح الرقمان الأول والثاني في المكانين الصحيحين، ثم تدخل العنصر الثالث (k=3) في المكان الصحيح، وهكذا إلى أن تدخل العنصر الأخير.



تدريب (6)

> لماذا استخدمنا في الخوارزمية أعلاه المتغير البوليني (inserted boolean variable) ؟

كما أسلفنا، عزيزي الدارس، فإن خوارزمية الفرز الإدخالي تبذل جهداً أقل إذا أعطيت مصفوفة مرتبة أو شبه مرتبة منذ البداية إذ أنها تقوم بعدد أقل من المقارنات والتبديلات في هذه الحالة. ومن هنا فإننا نحتاج إلى تحليل هذه الخوارزمية آخذين بعين الاعتبار سيناريوهات مختلفة مثل أفضل حالة (best case)، وأسوأ حالة (worst case).

يتضح مما سلف أن أفضل حالة لهذه الخوارزمية هي عندما تكون المصفوفة مرتبة أصلاً ولنحلل معاً هذه الحالة آخذين بعين الاعتبار عدد المقارنات اللازمة إذ أن عملية المقارنة هي العملية الحرجة في هذه الخوارزمية أيضاً.

إن عدد المقارنات اللازمة لإدخال العنصر الثاني هي مقارنة واحدة، وهو نفس العدد اللازم لإدخال العنصر الثالث والرابع إلخ. أي أن مجموع عدد عمليات المقارنة هو N-1 وبذلك تكون الخوارزمية O(N) في الحالة الأفضل.

يتضح مما سبق أيضاً أن الخوارزمية تقوم بأكبر عدد من المقارنات عندما تكون

المصفوفة مرتبة ترتيباً عكسياً (تنازلياً) إذ تشكل هذه الحالة الأسوأ. إذ تحتاج الخوارزمية عملية مقارنة واحدة لإدخال العنصر الثاني وإلى عمليتي مقارنة لإدخال العنصر الثالث وإلى ثلاث عمليات لإدخال العنصر الرابع وهكذا. يتضح مما تقدم أن عدد عمليات المقارنة اللازمة للحالة الأسوأ هو المجموع:

$$1 + 2 + 3 + ... + N - 1$$

$$\frac{1}{2}(N^2 - N)$$

وبذلك تكون الخوارزمية O(N2) في أسوأ الأحوال.

لنأخذ الآن عملية التبديل بعين الاعتبار حيث أنها قد تحتاج إلى وقت تنفيذي أكبر من عملية المقارنة، كما سلف وذكرنا، ولنعد تحليل خوارزمية الفرز الإدخالي.

إن عملية التبديل لن تنفذ على الإطلاق إذا كانت المصفوفة مرتبة، وستنفذ بعدد عمليات المقارنة في أسوأ الأحوال وهكذا تكون O(N2) أيضاً.

ونختتم هذه الوحدة بالقول بأن ما عرضناه فيها من طرق لتحليل الوقت اللازم لتنفيذ الخوارزميات باستخدام ترميز O الكبيرة يستخدم بالطريقة نفسها لتحليل كمية الذاكرة اللازمة لتنفيذ الخوارزميات.

أسئلة التقويم الذاتي (3)



عدد خوارزميات الفرز، وقارن الفروقات فيما بينها.

6. الخلاصة

إن تحليل الخوارزميات وتركيب البيانات موضوعان مهمان ومتلازمان. إذ أنه وللمقارنة بين تراكيب البيانات المختلفة لاختيار الأنسب منها لا بد من دراسة وتحليل الخوارزميات اللازمة لمعالجتها من حيث وقت وذاكرة الحاسوب اللازمين لتنفيذ الخوارزمية.

لتحليل الخوارزمية نحن لا نلجأ عادة إلى قياس الوقت الحقيقي اللازم لتنفيذ الخوارزمية باستخدام جهاز حاسوب معين، وذلك لأن الوقت الحقيقي قد يتأثر بعوامل ليس لها علاقة بجودة الخوارزمية كسرعة جهاز الحاسوب المستخدم وحجم البيانات المستخدمة في التجربة وطريقة تنفيذ الخوارزمية (تحويلها إلى برنامج). لذلك فإننا نلجأ إلى طريقة رياضية رمزية للتنبؤ بمقدار الوقت اللازم. لا شك أن وقت التنفيذ يعتمد على عدد خطوات الخوارزمية اللازم تنفيذها، والتي هي بدورها تعتمد على حجم البيانات N . ولتحديد هذا العدد بشكل تقريبي نحدد العملية الحرجة في الخوارزمية، وهي العملية الأكثر تكراراً عند التنفيذ، ثم نعبر عن هذا العدد مستخدمين اقترانا T(N) بدلالة N. ثم نحاول إيجاد الاقتران F(N) الذي يحقق الشرط التالي:

$$T(n) <= c* F(n)$$

$$n >= m$$

حيث m وc هما ثابتان عدديان، عندها نستطيع القول بأن الخوارزمية هي O(F(n)) والهدف من هذه الطريقة والتي تدعى بطريقة ترميز O الكبيرة (Big-O notation) هو التخلص من العوامل غير المهمة في الاقتران T(N). قد يعتمد عدد خطوات الخوارزمية على طبيعة البيانات المعالجة وفي هذه الحالة يكون للخوارزمية حالة تدعى الحالة الأفضل، وهي الحالة التي تتطلب تنفيذ أقل عدد ممكن من الخطوات وحالة تدعى بالحالة الأسوأ وهي الحالة التي تتطلب تنفيذ أكبر عدد من الخطوات، وحالة أخيرة تدعى بالحالة الوسطى.

واختتمنا الوحدة، عزيزي الدارس، بعرض وتحليل ثلاث خوارزميات لفرز البيانات التي هي خوارزمية الفرز الفقاعي، وخوارزمية الفرز الانتقائي، وخوارزمية الفرز الإدخالي.

7. لمحة عن الوحدة الدراسية الثالثة

في الوحدة التالية، عزيزي الدارس، سوف ندرس منهجية لتصميم تراكيب البيانات وتنفيذها كتراكيب بيانات تجريدية بحيث يستطيع المرء استخدامها في برامجه دون الحاجة إلى معرفة تفاصيل تنفيذ هذه التراكيب. وفيما لو اضطررنا إلى تغيير تفاصيل هذا التنفيذ فإننا لا نضطر إلى تغيير البرامج المستخدمة لهذه التراكيب البيانية.

8. إجابات التدريبات

تدريب (1)

بما أن هذين الدورانين متداخلين، فإن العمليات الحرجة هي العمليات داخل الدورانين. ولكون الدوران الداخلي يتكرر N/2 من المرات والدوران الخارجي يتكرر N2 من المرات فإن عدد مرات تنفيذ العمليات الحرجة هو N/2*N2 وعليه فإن الخوارزمية هي O(N3).

تدريب (2)

واضح في هذه المسألة أن العملية الحرجة هي عملية الكتابة cout وتتكرر 300 مرة، وهكذا تكون الخوارزمية O(1) وذلك لأن

C*1=>300

حيث C=300.

تدريب (3)

لاحظ، عزيزي الدارس، أن الدوران الداخلي يتكررlog2N من المرات، وأن الدوران الخارجي يتكررN من المرات، وعليه فإن العملية الحرجة داخل الدورانين تتكرر $N*\log_2 N$ وعليه تكون الخوارزمية $(N*\log_2 N)$.

تدريب (4)

لاحظ، عزيزي الدارس، أن الدورانين في هذه الخوارزمية ليسا متداخلين، بل هما متتابعان بمعنى أن الدوران الثاني يبدأ عندما ينتهي الأول. وبما أن الدوران الثاني يتكرر عدداً أكبر من المرات فإن العملية الحرجة هي العملية التي تقع داخل هذا الدوران وتتكرر N^2 من المرات وعليه تكون الخوارزمية $O(N^2)$.

تدريب (5)

ذلك لأنك لو وضعت N-1 من العناصر في المكان الصحيح فإن العنصر الأخير سيكون حتماً في المكان الصحيح.

تدريب (6)

وذلك حتى تتوقف عملية المقارنة (الدوران الداخلي) عندما يصل العنصر إلى موقعه الصحيح في المصفوفة.

9. مسرد المصطلحات

- **تحليل الحالة الأسوأ Worst Case Analysis**: الحالة التي تحتاج فيها الخوارزمية إلى تنفيذ أكبر عدد من الخطوات.

- **تحليل الحالة الأفضل Best Case Analysis**: الحالة التي تحتاج فيها الخوارزمية إلى تنفيذ أقل عدد من الخطوات.

- **تحليل الحالة الأوسط Average Case Analysis**: الحالة التي تتطلب من الخوارزمية تنفيذ عدد معتدل من الخطوات.

- **تحليل الخوارزميات Algorithm Analysis**: دراسة الخوارزمية والتعرف على مدى كفاءتها من حيث وقت التنفيذ اللازم وذاكرة الحاسوب اللازمة.

- **ترميز O الكبيرة Big-O Notation**: طريقة رمزية (notation) تختص في استخلاص العوامل المهمة في التعبير عن كفاءة الخوارزمية.

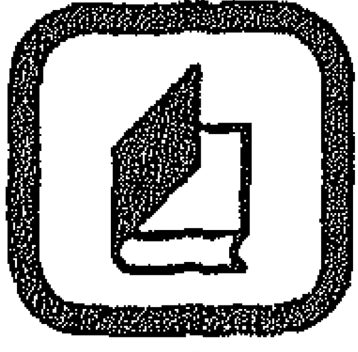
- **خوارزميات الترتيب Sort Algorithm**: مجموعة من الخوارزميات لترتيب قوائم (lists) من البيانات، حيث أنها تعمل على ترتيبها وفق معايير معينة وفي الغالب ترتيبها ترتيباً تصاعدياً أو تنازلياً.

- **خوارزمية Algorithm**: مجموعة من الخطوات الممنهجة تتسم بسمات محددة تؤدي إلى تحقيق هدف ما أو مجموعة أهداف محددة.

- **خوارزمية الفرز الإدخالي The Insertion Sort Algorithm**: تعمل على ترتيب العناصر (ابتداء من العنصر الثاني) بإدخال كل عنصر في المكان الصحيح نسبة إلى جميع العناصر التي سبق ترتيبها (إدخالها).

- **خوارزمية الفرز الانتقائي The Selection Sort Algorithm**: تعمل الخوارزمية على ترتيب عناصر المصفوفة بتكرار عمليتين إيجاد أقل عنصر في الجزء غير المرتب من المصفوفة واستبدال ذلك العنصر مع أول عنصر في الجزء غير المرتب.

- **خوارزمية الفرز الفقاعي The Bubble Sort Algorithm**: خوارزمية بسيطة تقوم بمقارنة كل عنصرين متجاورين في المصفوفة، واستبدالهما إذا وجد أن العنصر الأول أكبر من الذي يليه.



10. المراجع

1. Tremplay, J.P.; and Sorenson, P. G.; An Introduction to Data Structures with Applications, 2nd Edition, McGraw-Hill, 1984.
2. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice- Hall. 2000.
3. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison- Wesley, 1999.

الوحدة الثالثة

التراكيب التجريدية

محتويات الوحدة

1. المقدمة

1.1 تمهيد

أهلاً بك، عزيزي الدارس، في الوحدة الثالثة، من كتاب "تركيب البيانات وتصميم الخوارزميات وهي بعنوان "التراكيب التجريدية" ستتعرف، عزيزي الدارس، في هذه الوحدة على أهمية إنشاء تراكيب البيانات كتراكيب تجريدية وكيفية عمل ذلك وفوائده. وهو موضوع في غاية الأهمية إذ أن استيعابك واستخدامك للتراكيب التجريدية في برامجك يجعلك أسرع وأكفأ في إنجاز مهامك البرمجية سواء كان ذلك في كتابة برامج جديدة أو في التعديل على برامج قديمة.

2.1 أهداف الوحدة

ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1. توضح مفهوم التراكيب التجريدية.
2. تذكر فوائد استخدام التراكيب التجريدية.
3. تعرف وتنفذ تراكيب بيانات جديدة.
4. تستخدم التراكيب التجريدية في برامجك التطبيقية بطريقة صحيحة.

3.1 أقسام الوحدة

تتكون هذه الوحدة من أربعة أقسام رئيسة ترتبط بقائمة الأهداف السابقة. في القسم الأول، سنناقش مفهوم التراكيب التجريدية وفوائدها وهو مرتبط بالهدفين الأول والثاني، وفي القسم الثاني سنناقش كيفية تعريف وتنفيذ تراكيب بيانات تجريدية وهو مرتبط بالهدف الثالث. والقسم الثالث يوضح مثالاً متكاملاً عن كيفية إنشاء تركيبة بيانات تجريدية جديدة وهو أيضاً مرتبط بالهدف الثالث. أما القسم الرابع فيوضح كيفية استخدام التراكيب التجريدية في برامج تطبيقية بطريقة صحيحة وهو مرتبط بالهدف الرابع.

4.1 القراءات المساعدة

حاول، عزيزي الدارس، الانتفاع بالمراجع التالية لارتباطها بموضوع هذه الوحدة:

1. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice -Hall, 2000.
2. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison Wesley, 1999.

2. التراكيب التجريدية

التراكيب التجريدية هي تراكيب البيانات التي تستطيع استخدامها من خلال عمليات معينة معرفة على شكل دوال على هذه التراكيب، من دون الحاجة إلى معرفة تفاصيل تمثيل هذه التراكيب. على سبيل المثال، الأرقام الصحيحة في لغة سي++ (Integer data type) هي نوع من أنواع البيانات التجريدية بالنسبة لنا إذ أننا نستخدم هذا النوع من خلال عمليات معرفة عليها مثل الجمع والضرب والقسمة والطرح، وبدون أن نضطر إلى معرفة كيفية تمثيل الأعداد الصحيحة فعلياً. إذ أننا لا نحتاج لأن نعرف هل كل عدد صحيح ممثل باستخدام بايت واحد أو أكثر؟ وما طريقة تمثيل الأعداد السالبة، ومع ذلك فنحن قادرون على استخدام الأعداد الصحيحة في لغة سي++ بكل سهولة ويسر.

وهذا ما ينبغي علينا عمله عندما نصمم وننشئ تركيبة بيانات جديدة، أي يجب علينـا أن ننشئهـا كتركيبـة بيانات تجريدية بحيث نستطيع (ويستطيع آخرون) استخدامها فيما بعد من خلال عمليات معرفـة عليها دون الحاجة إلى معرفة تفاصيل تمثيلها وتنفيذها. ونهدف بذلك إلى إخفاء الكثير من المعلومات (Information Hiding)، المتعلقة بكيفية تمثيل وتنفيذ تركيبة البيانات. على سبيل المثال، لو أردنا إنشاء تركيبة بيانات لتمثيل قائمة من المعلومات الخاصة بطلبة لنطلق عليها اسم Student List فيجب أن نصممها وننشئها بحيث نستطيع (ويستطيع زملاؤنا) استخدامها دون الحاجة إلى معرفة تفاصيل تمثيـل هذه القائمة، فقد تمثل القائمة كمصفـوفة من السجـلات (Array of Structure) أو كقائمة متصلة (Linked List)، أو بطرق أخرى. كي تكون تركيبة البيانات هذه تركيبة تجريدية يجب أن تستطيع استخدامها دون الحاجة إلى معرفة هذه التفاصيل، كل ما تحتاج إلى معرفته كي تستطيع استخدام تركيبة البيانات هذه هو العمليات (Operations) المُعّرفة عليها وكيفية استدعاء هذه العمليات. في مثالنا هذا قد تتكون مجموعة العمليات هذه من: عملية لإدخال طالب جديد في القائمة List Insert، وعملية للبحث عن طالب معين List Search، وأخرى لحذف طالب من القائمة List Delete، وأخرى لطباعة القائمة List Display وهكذا.

وعادة ما تنفذ هذه العمليات كدوال (Functions). وبالطبع يجب أن نكون قادرين على استدعاء الدوال دونما الحاجة إلى معرفة تفاصيل محتوياتها وخطوات عملها، كل ما نحتاج إلى معرفته كي نستطيع استدعاء الدوال هو اسم الدالة ونوع ومعاملات (Arguments) هذه الدالة ونوع البيانات التي ترجعها الدالة.

ولا بد أنك تتساءل، عزيزي الدارس، ما الفائدة التي نجنيها من عمل ذلك؟ أي ما الفائدة التي نجنيها من مقدرتنا على التعامل مع التراكيب البيانية بدون الحاجة إلى معرفة تفاصيل تمثيلها وتنفيذها؟ هنالك، عزيزي الدارس، عدة فوائد لذلك منها أن المستخدم وهو قد يكون مبرمجاً آخر، يحاول استخدام تركيبة البيانات هذه في برامج تطبيقية كبيرة ومعقدة، يستطيع أن يركز جهده وتفكيره على المسألة التي يحاول كتابة برنامج لها بدون الحاجة إلى التفكير في الوقت نفسه في تفاصيل تمثيل تركيبة البيانات، ولذلك فهو يصبح أكثر إنتاجاً وفاعلية، وأيضاً تصبح برامجه أقل تعقيداً وأيسر للكتابة والفهم والتعديل.

تصور، عزيزي الدارس، أنك بحاجة إلى معرفة طريقة تمثيل الأعداد الصحيحة السالبة منها والموجبة كي تستطيع جمع رقمين من هذه الأرقام، وطلب إليك أن تكتب برنامجاً بسيطاً لجمع الأرقام من 1 إلى 50. لا شك أن مهمتك لن تكون سهلة وستأخذ وقتاً طويلاً لكتابة البرنامج الذي سيكون معقداً ومليئاً بتفاصيل تمثيل الأعداد الموجبة.

والفائدة الثانية والتي لا تقل أهمية، عزيزي الدارس، أنه في حالة تغيير طريقة تمثيل وتنفيذ تركيبة بيانات معينة فإن البرامج التطبيقية الكبرى التي تستخدم تركيبة البيانات هذه لن تتأثر أي أنها لن تحتاج إلى عمل تعديلات عليها طالما أن العمليات المُعّرفة أصلاً على تركيبة البيانات ما زالت مُعّرفة، وأن طريقة استخدامها (استدعائها) لم تتغير. بالطبع قد تكون خطوات عمل الدوال المقابلة لهذه العمليات قد تغيرت، ولكن المهم أن تبقى طريقة استدعائها بدون تغيير.

لنعد إلى مثالنا السابق لنفرض أننا مثلنا ونفذنا قائمة الطلبة كمصفوفة ذات بعد واحد وبعد فترة من الزمن وجدنا أن هذه الطريقة ليست فعّالة من حيث وقت التنفيذ اللازم للعمليات، أو الذاكرة اللازمة وأردنا تمثيل هذه القائمة كقائمة متصلة وليس كمصفوفة، إذا كنا وزملاؤنا قد استخدمنا منذ البداية تركيبة البيانات هذه كتركيبة بيانات تجريدية (من خلال العمليات المعرفة عليها فقط بدون الاعتماد على طريقة تمثيلها كمصفوفة) في برامجنا، عندها لن نضطر إلى تغيير هذه البرامج عند تغيير طريقة تنفيذ تركيبة البيانات هذه إلى قائمة متصلة. كل ما نحتاج إلى تغييره هو الدوال الممثلة للعمليات المعرفة فقط. وليس البرامج التطبيقية الأخرى المستخدمة لتركيبة البيانات هذه كبرنامج تسجيل الطلبة على سبيل المثال.

الفائدة الثالثة هي ما يسمى بالاستخدام المتكرر للبرامج (Code Reuse) بدون الحاجة إلى كتابتها مرة أخرى. تصور، عزيزي الدارس، أننا احتجنا إلى استخدام قائمة من الطلبة في برنامج جديد نحن بصدد تطويره لإيجاد بعض المعلومات الإحصائية عن الطلبة المسجلين في الجامعة، عندها نستطيع استخدام تركيبة البيانات Student List بدون الحاجة إلى إعادة تمثيلها وتنفيذ العمليات المُعّرفة عليها (كتابة الدوال مرة أخرى) من إدخال طالب إلى القائمة أو حذف طالب وما إلى ذلك. وهذا لا يختصر الوقت فقط بل يزيد مقدار ثقتنا في برنامجنا الجديد، لأنه يستخدم دوال سبق استخدامها والتأكد من صحتها.

الفائدة الرابعة لاستخدام التراكيب التجريدية هي تسهيل عمل فرق التصميم والبرمجة. تعلم، عزيزي الدارس، أن البرامج التطبيقية الكبيرة تُطور في الغالب من قبل فرق عمل من محللين ومبرمجين. إن استخدام تراكيب بيانات تجريدية موحدة في البرامج الفرعية المختلفة التي يكتبونها يساعد على جعلها متوافقة بحيث يستطيع برنامج فرعي معين استدعاء برنامج فرعي آخر، إذ أنهما يستخدمان تركيبة بيانات واحدة مُمثلة ومُنفذة بطريقة واحدة.



أسئلة التقويم الذاتي (1)

1. عرّف ما المقصود بتراكيب البيانات التجريدية.
2. وضح الفوائد المختلفة لاستخدام تراكيب البيانات التجريدية.

3. تصميم التراكيب التجريدية وتنفيذها

تعلم، عزيزي الدارس، أن المهندس المسؤول عن إنشاء أي جسر معين على نهر، يقوم بتنفيذ مهمته على مراحل ومروراً بخطوات لا يستطيع تجاوز أي منها فهو يقوم بفهم ودراسة متطلبات واحتياجات الجهة المؤسسة لهذا الجسر، ثم ينتقل إلى مرحلة التصميم والتخطيط، وفي المرحلة الأخيرة يقوم بعملية التنفيذ والإنشاء. كذلك هنالك، عزيزي الدارس، خطوات لتصميم وتمثيل التراكيب التجريدية يجب الالتزام بها وعدم تجاوز أي منها لضمان تصميم وتنفيذ تركيبة البيانات بأفضل طريقة ممكنة وحتى يتحقق الهدف المرجو منها.

لإنشاء تركيبة تجريدية جديدة علينا المرور بمراحل شبيهة بتلك التي يمر بها ذلك المهندس. إذ في المرحلة الأولى وهي مرحلة التعريف نعرف بدقة تركيبة البيانات الجديدة ونحدد مواصفاتها قبل أن نفكر بتفاصيل البناء (التمثيل)، وفي المرحلة الثانية، وهي مرحلة التمثيل، نحدد تفاصيل التمثيل المناسبة كاستخدام المصفوفات أو القوائم المتصلة وما إلى ذلك.

ولتعريف تركيبة البيانات هنالك ثلاثة عوامل يجب أخذها بعين الاعتبار:

1. أن نعطي وصفاً للعناصر المكونة لتركيبة البيانات.
2. أن نعطي وصفاً للعلاقة بين هذه العناصر.
3. نعطي وصفاً للعمليات التي نرغب بتنفيذها على تركيبة البيانات، وهذا الوصف عادة ما يكون عبارة عن ترويسة الدوال (Function header) والدوال المنفذة لهذه التعليمات بلغة سي++ على سبيل المثال، لا نقصد هنا كتابة الدالة كاملة بل فقط ترويسة تلك الدالة (أول سطر في الدالة) كما سنرى.

تذكر، عزيزي الدارس، أن هذه العوامل الثلاثة أسميناها في الوحدة الأولى بالتصور المنطقي لتركيبة البيانات، وهي نظرة نابعة من الوظيفة التي تؤديها هذه البيانات، ولا بد في نهاية المطاف من تمثيل هذا التصور المنطقي بطريقة مناسبة باستخدام لغة برمجة معينة لتحويلها إلى ما أسميناه في الوحدة الأولى بالتصور الفيزيائي لتركيبة البيانات وهي نظرة نابعة من طبيعة تكوين الجهاز وطريقة تعامله مع البيانات من حيث التخزين.

إن تعريفنا لتركيبة البيانات أي وصفنا للتركيب المنطقي في هذه المرحلة يجب أن يحتوي كل المعلومات اللازمة لاستخدام هذه التركيبة من قبل أشخاص آخرين دون الحاجة

إلى معرفة تفاصيل التمثيل. وبعد مرحلة التعريف تأتي مرحلة التمثيل حيث يتقرر في هذه المرحلة كيفية تمثيل تركيبة البيانات باستخدام مصفوفة ذات بعد واحد على سبيل المثال أو باستخدام مصفوفة ذات بعدين أو باستخدام قائمة متصلة أو بطريقة أخرى مختلفة. وبناء على ذلك يتم تجهيز أو كتابة الدوال والدوال اللازمة لتنفيذ العمليات التي حددت في مرحلة التعريف، والسبب في تأخير هذه المرحلة (مرحلة التمثيل) إلى ما بعد مرحلة التعريف هو كي تصبح في وضع أفضل لتقرير كيفية تمثيل البيانات بعد أن فهمنا مكوناتها وكيفية استخدامها من خلال العمليات المعرفة عليها.



أسئلة التقويم الذاتي (2)

1. ما العوامل الواجب أخذها بعين الاعتبار عند تعريف تركيبة بيانات تجريدية ؟
2. ما الحكمة من تأخير مرحلة تراكيب البيانات التجريدية إلى ما بعد مرحلة التعريف؟

4. المجموعة باعتبارها تركيبة بيانات تجريدية
The Set Abstract Data Type

في هذا القسم، عزيزي الدارس، سنعرض مثالاً توضيحياً لكيفية إنشاء تركيبة بيانات تجريدية لتمثيل المجموعات سنطلق عليها اسم (setType) وسنوضح كيفية استخدامها في برنامج تطبيقي. يتضح مما سبق، عزيزي الدارس، أنه وقبل التفكير في كيفية تمثيل المجموعة يجب أن نُعرف ما نقصده بكلمة المجموعة ونُعرف العمليات التي نود تنفيذها على المجموعات.

1.4 تعريف المجموعة باعتبارها تركيبة بيانات تجريدية setType

المجموعة هي عدد من الأشياء غير المكررة المأخوذة من مدى (Universe) محدد من الأشياء. ولا يشترط أن تكون عناصر المجموعة مرتبة بأي ترتيب معين والعمليات التي تنفذ عادة على المجموعات هي:

1. إنشاء المجموعة SetCreate:

حيث تنشأ المجموعة كمجموعة خالية (Empty Set)، ويجب استخدام هذه التعليمة قبل استخدام المجموعة لأول مرة، وتحتاج في هذه المرحلة (مرحلة التعريف) إلى كتابة ترويسة الدالة التي ستنفذ هذه العملية فقط. والترويسة هي:

```
SetCreate(SetType& S);
```

2. الانتماء إلى المجموعة IsElementof:

حيث تقرر هذه العملية فيما إذا كان عنصر ما E منتمياً إلى المجموعة S أم لا.

لاحظ، عزيزي الدارس، أن العنصرE لا بد أن يكون من مدى (Universe) المجموعة، أي من نفس نوع الأشياء المسموح لها الانتماء إلى المجموعة، لاحظ أيضاً أن نتيجة العملية والتي سننفذها كدالة يعيد لنا القيمة البولينية true إذا كان العنصر منتمياً إلى المجموعة ويعيد لنا القيمة البولينية false إذا لم يكن منتمياً إلى المجموعة. أي أن الدالة هي دالة بولينية Boolean Function، وعليه تكون ترويسة الدالة كما يلي:

```
bool IsElementOf(setType S, int E);
```

3. تعيين المجموعات SetAssign

تعين هذه العملية قيمة المجموعة S إلى مجموعة أخرى T أي أنها تجعلهما متساويتين

```
void SetAssign(setType S,  setType& T);
```

4. التأكد من أن المجموعة خالية SetEmpty

وهي عملية نستخدمها إذا أردنا فحص مجموعة E هل هي خالية أم لا. مرة أخرى دالة بولينية Boolean function يعيد لنا القيمة true إذا كانت المجموعة خالية وfalse إذا لم تكن خالية سيكون مناسباً لتنفيذ هذه العملية

```
bool SetEmpty(setType s);
```

5. التأكد من مساواة المجموعات SetEqual

وتستخدم لفحص فيما إذا كانت المجموعتان S, T متساويتين

```
bool SetEqual(setType S, setType T);
```

6. فحص المجموعة الجزئية SubSetOf

وتستخدم لفحص فيما إذا كانت المجموعة S هي مجموعة جزئية في المجموعة T

```
bool SubSetOf(setType S,setType T);
```

7. اتحاد المجموعات Union

وتستخدم لإيجاد المجموعة T وهي مساوية لاتحاد المجموعتين R وS.

```
void Union(setType S,setType R, setType& T);
```

8. تقاطع المجموعات Intersection

وتستخدم هذه العملية لإيجاد المجموعة T وهي مكونة من العناصر الموجودة في كل من المجموعتين R وS.

```
void Intersection(setType S,setType R, setType& T);
```

9. فرق المجموعات Difference

وتستخدم هذه العملية لإيجاد المجموعة T المكونة من العناصر الموجودة في المجموعة R وليست موجودة في المجموعة S.

```
void Difference(setType S,setType R, setType& T);
```

10. إضافة عنصر إلى المجموعة AddElement

وتستخدم هذه العملية لإضافة العنصر U من المدى Universe إلى المجموعة S.

```
void AddElement(setType& S,int U);
```

11. إزالة عنصر من المجموعة RemoveElement

وتستخدم لإزالة العنصر U من المجموعة S.

```
void RemoveElement(setType& S,int U);
```

12. طباعة عناصر المجموعة SetDisplay

تقدم هذه الدالة بعرض العناصر المنتمية إلى المجموعة على شاشة العرض.

```
void SetDisplay(setType S);
```

لاحظ، عزيزي الدارس، أن المعلومات الواردة أعلاه لتعريف تركيبة بيانات المجموعة setType هي تقريباً ما يلزم معرفته لاستخدامها بفاعلية بدون الحاجة إلى معرفة تفاصيل التمثيل والتنفيذ. كل ما يحتاج أي مستخدم لعمله هو الحصول على نسخة من البرامج الفرعية المنفذة للعمليات واستخدامها بدون الحاجة إلى دراستها ومعرفة تفاصيلها. بالطبع سوف يكون هذا المستخدم بحاجة إلى تعريف مدى المجموعة Universe (الأشياء المكونة للمجموعة Universe) وهو قد يكون مجموعة من أيام الأسبوع أو أشهر السنة أو أسماء الحيوانات وما إلى ذلك، بالطبع فإن طبيعة البرنامج المستخدم للمجموعات هي التي تملي ماهية المدى Universe.

2.4 تنفيذ المجموعات باعتبارها تركيبة بيانات تجريدية setType

الآن، عزيزي الدارس، وبعد أن عرفنا المجموعات والكيفية التي ستستخدم بها نستطيع الانتقال إلى مرحلة التنفيذ. حيث نبحث في كيفية تمثيل المجموعة وكيفية كتابة الدوال والدوال المنفذة للعمليات المعرفة أعلاه.

تذكر، عزيزي الدارس، أن المجموعة set كنوع من أنواع البيانات هي موجودة أصلا في لغة سي++ وهنالك العديد من العمليات المعرفة عليها، ولكننا سنتجاهل ذلك، وننفذ المجموعات باعتبارها تركيبة بيانات جديدة، وذلك بهدف توضيح كيفية التعامل مع تركيبات البيانات التجريدية.

سنفترض هنا أن مدى المجموعة universe هو نوع من أنواع البيانات الترتيبية Ordinal data type. تذكر، عزيزي الدارس، أن البيانات الترتيبية هي ذلك النوع من البيانات الذي يكون لكل عنصر فيه، باستثناء أول عنصر وآخر عنصر، عنصر سابق Predecessor وعنصر لاحق Successor. على سبيل المثال الأعداد الصحيحة كنوع من أنواع البيانات هو نوع ترتيبي حيث لكل رقم صحيح رقم سابق ورقم لاحق. كذلك الأمر

بالنسبة لنوع البيانات الرمزية (Char)، حيث لكل رمز هناك رمز سابق ورمز لاحق، على سبيل المثال الرمز 'C' الرمز السابق له هو 'B' والرمز اللاحق له هو'D'. ولكن الأعداد الحقيقية لا تعتبر من أنواع البيانات الترتيبية (Ordinal) وذلك لعدم وجود رقم سابق ورقم لاحق متفق عليه لأي رقم على سبيل المثال الرقم 1.1، ما الرقم السابق له؟ هل هو 1.0، أم 1.90، أم هل هو 1.009، أم ماذا؟

ما يهمنا هنا أن يكون مدى المجموعة universe من نوع ترتيبي، وذلك لأننا سنمثل المجموعة مستخدمين مصفوفة ذات بعد واحد، كل موقع في المصفوفة يأخذ القيمة true أوfalse أي أننا سنستخدم مصفوفة بولينيةBoolean Array . سنخصص في المصفوفة موقعاً لكل قيمة من قيم المدى الترتيبي ستكون القيمة المخزنة في الموقع true إذا كانت قيمة المدى المقابلة عنصراً في المجموعة، وستكون القيمة المخزنة في الموقع false إذا لم تكن قيمة المدى المقابلة عنصراً في المجموعة.

على سبيل المثال لنفرض أن مدى المجموعة هو مجموعة الأحرف الإنجليزية 'A','B',............ 'Z' لتمثيل هذه المجموعة نحتاج إلى مصفوفة ذات بعد واحد من 26 موقعاً وهو عدد الأحرف الإنجليزية لتخزين قيم بولينية (true أو false).

الآن للتعبير عن مجموعة الأحرف: **{A, E, I, O, U}**

نستخدم المصفوفة ذات البعد الواحد التالية:

A	B	C	D	E	F	G	H	I	J	K
true	false	false	false	true	false	false	false	true	false	false

L	M	N	O	P	Q	R	S	T	U	V
false	false	false	true	false	false	false	false	false	true	false

W	X	Y	Z
false	false	false	false

لاحظ، عزيزي الدارس، أن في المصفوفة موقعاً مخصصاً لكل قيمة محتملة من قيم مدى المجموعة (لكل حرف من الأحرف الإنجليزية)، لكل حرف موجود في المجموعة خزن في الموقع المقابل في المصفوفة true ولكل حرف ليس عنصراً في المجموعة خزن من الموقع المقابل في المصفوفة false.

وعليه سنمثل المجموعة في القسم الخاص من الصنف setType في برنامج سي++ كما يلي:

```
bool set[universe];
```

حيث firstValue و lastValue ثابتان يحددان من قبل المستخدم، في مثالنا firstValue هي 'A' وlastValue هي 'Z'.

لننتقل الآن إلى موضوع تنفيذ العمليات المختلفة التي حددناها في مرحلة التعريف ومن المهم جداً، عزيزي الدارس، أن نتقيد هنا بترويسة الدوال (function headers) التي حددناها في مرحلة التعريف، وذلك لأننا نريد أن يستطيع المرء استخدام تركيبة البيانات فقط من خلال معرفة تلك المعلومات التي حددناها في مرحلة التعريف، فلا يجوز أن نكتب الدالة الخاصة بعملية اتحاد المجموعات مفترضين أنه يأخذ معاملاً جديداً (لم يكن موجوداً في ترويسة الدالة) كعدد العناصر في المجموعة على سبيل المثال.

1. إنشاء المجموعة SetCreate:

كما اتفق في مرحلة التعريف ينشئ هذه الدالة مجموعة خالية، وعليه يجب أن يخزن القيمة false في كل موقع من مواقع المصفوفة.

```
void setType:: SetCreate(setType&  S)
  {for(int i=firstValue;i<=lastValue;i++)
   S.set[i]=false;
   false;
  } // end  SetCreate
```

2. الانتماء إلى المجموعة IsElementOf

حيث تعيد هذه الدالة القيمة البولينية true إذا كان العنصر E منتمياً إلى المجموعة S والقيمة false إذا لم يكن عنصراً في المجموعة.

لاحظ، عزيزي الدارس، أن هذه الدالة يجب أن تعيد القيمة البولينية المخزنة في موقع المصفوفة المقابل للعنصر E، وعليه فإننا نستطيع كتابة ما يلي:

```
bool setType::IsElementOf(setType S, int E)
 { return S.set[E];
 }//  end iselement
```

3. تعيين المجموعات SetAssign

تجعل هذه الدالة قيمة المجموعة T مساوية لقيمة المجموعة S.

```
void setType::SetAssign(setType S,  setType& T)
 {
  for(int i=firstValue;i<=lastValue;i++)
  T.set[i]=S.set[i];
 }// end SetAssign
```

4. التأكد من أن المجموعة خالية SetEmpty

تكون المجموعة خالية إذا كانت كل القيم المخزنة في المصفوفة هي false وإلا فإنها لا تكون خالية وعليه يجب أن تعيد الدالة القيمة false في هذه الحالة.

```
bool setType::SetEmpty(setType S)
 {
 for(int i=firstValue;i<=lastValue;i++)
  if (S.set[i]==true)
   return false;
 }// end SetEmpty
```

5. التأكد من مساواة المجموعات SetEqual

سنترك لك، عزيزي الدارس، كتابة هذه الدالة كتدريب.



تدريب (1)

اكتب دالة بولينية لفحص فيما إذا كانت المجموعتان S, T متساويتين.

6. فحص المجموعة الجزئية SubsetOf

تكون المجموعة S مجموعة جزئية في T إذا كان كل موقع في المصفوفة الممثلة للمجموعة S يحتوي على القيمة true وكان الموقع المقابل نفسه في المصفوفة الممثلة للمجموعة T محتوياً على القيمة true أيضاً.

```
bool setType::SubSetOf(setType S,setType T)
 {
 for(int i=firstValue;i<=lastValue;i++)
   if (S.set[i] && !T.set[i])  return false;
 }
```

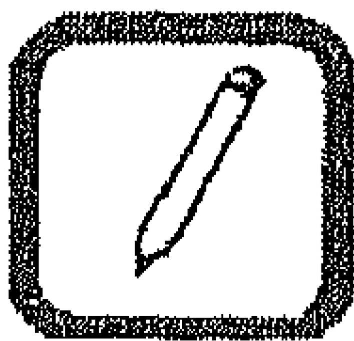
7. اتحاد المجموعات Union

لإيجاد المجموعةT التي هي حاصل اتحاد المجموعتين R وS علينا أن نضع فيها كل عنصر موجود في المجموعة R أو موجود في المجموعة S.

```
void setType::Union(setType S,setType R, setType& T)
 {
  for(int i=firstValue;i<=lastValue;i++)
  T.set[i]=R.set[i]|| S.set[i];
  } //end union
```

8. تقاطع المجموعات Intesection

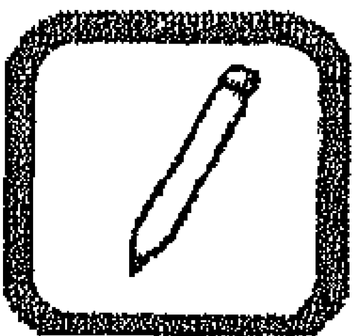
وسنترك لك، عزيزي الدارس، كتابة هذه الدالة والدوال التالية كتدريب.



تدريب (2)

اكتب دالة لإيجاد حاصل تقاطع مجموعتين، استخدم نفس ترويسة الدالة (Function Heading) التي حددناها في مرحلة التعريف.

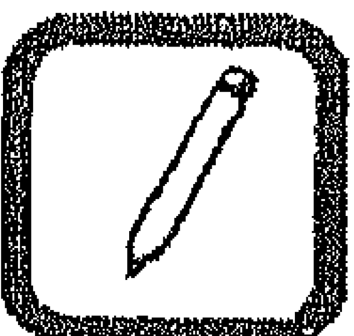
9. فرق المجموعات Difference



تدريب (3)

اكتب دالة لإيجاد حاصل طرح مجموعتين، استخدم نفس ترويسة الدالة (Function Heading) التي حددناها في مرحلة التعريف.

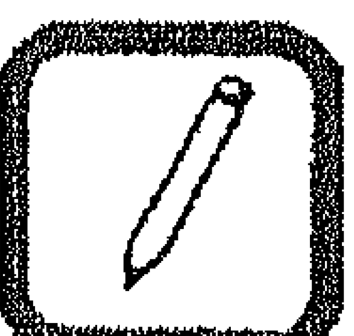
10. إضافة عنصر إلى المجموعة AddElement



تدريب (4)

اكتب دالة لإضافة عنصر جديد إلى مجموعة ما، استخدم نفس ترويسة الدالة المحددة في مرحلة التعريف.

11. حذف عنصر من المجموعة RemoveElement



تدريب (5)

اكتب دالة لحذف عنصر ما من مجموعة ما، استخدم نفس ترويسة الدالة المحددة في مرحلة التعريف.

12. طباعة عناصر المجموعة SetDisplay

تقوم هذه الدالة بعرض العناصر المنتمية إلى المجموعة على شاشة العرض.

إن كل ما تفعله هذه الدالة هو فحص قيم المدى Universe على الترتيب؛ ليحدد أياً منها تنتمي إلى المجموعة، وتقوم بعرضها على الشاشة.

```
void setType::SetDisplay(setType S)
 {for(int i=firstValue;i<=lastValue;i++)
   cout<<S.set[i]<<» «;
 }
```



أسئلة التقويم الذاتي (3)

ما الحكمة من التقيد بترويسة العمليات المحددة في مرحلة التعريف عند تنفيذ هذه العمليات؟

5. استخدام التراكيب التجريدية

في هذا القسم، عزيزي الدارس، سنناقش كيفية استخدام التراكيب التجريدية في برنامج تطبيقي بدون الحاجة إلى معرفة تفاصيل تمثيل وتنفيذ هذه التراكيب. إذ سنكتب برنامجاً يقرأ سطراً من الأحرف الإنجليزية (line of text) ثم يحدد مجموعة أحرف العلة (vowels) المستخدمة في هذا السطر، ثم يقوم بإظهار عناصر هذه المجموعة على شاشة العرض.

تذكر، عزيزي الدارس، أن أحرف العلة في اللغة الإنجليزية هي A وE وI وO وU بحالتيها الكبيرة(uppercase) والصغيرة (lowercase). وبهذا يكون مدى universe المجموعات التي ستستخدم في برنامجنا هو الأحرف الكبيرة والصغيرة أي أن قيمتي الثابتين firstValue وlastValue هما 'A' و'z' (لاحظ 'z' حرف صغير lowercase).

سنستخدم في برنامجنا أربع مجموعات:

1. LowerVowels وتستخدم لتخزين أحرف العلة الصغيرة (a وe وi وo وu).
2. UpperVowels وتستخدم لتخزين أحرف العلة الكبيرة (A وE وI وO وU).
3. Vowels وهي حاصل اتحاد LowerVowels و UpperVowels.
4. InLineVowels وتستخدم لتخزين أحرف العلة التي نجدها في سطر الأحرف المدخل.

بعد إنشاء المجموعات السابقة يقوم البرنامج بقراءة السطر حرفاً حرفاً (character by character) ثم يفحص فيما إذا كان الحرف عنصراً في المجموعة Vowels إن حدث ذلك فإن الحرف يضاف إلى مجموعة أحرف العلة المستخدمة في السطر InLineVowels.

وفيما يلي نص البرنامج (1):

```
#include <iostream.h>
#include <conio.h>
char firstValue=>A>, lastValue=>z>;
const universe=128;
class setType{
      private:    bool set [universe];
  public:
  void SetCreate(setType&  S);
  bool IsElementOf(setType S, int E);
  void SetAssign(setType S,  setType& T);
  bool SetEmpty(setType s);
  bool SetEqual(setType s);
  bool SubSetOf(setType S,setType T);
  void Union(setType S,setType R, setType& T);
  void Intersection(setType S,setType R, setType& T);
  void Difference(setType S,setType R, setType& T);
  void AddElement(setType& S,int U);
  void RemoveElement(setType& S,int U);
  void SetDisplay(setType S);
  };
void setType:: SetCreate(setType&  S)
   {for(int i=firstValue;i<=lastValue;i++)
    S.set[i]=false;}
bool setType::IsElementOf(setType S, int E) {  return S.set[E]; }
void setType::SetAssign(setType S,  setType& T)
 { for(int i=firstValue;i<=lastValue;i++)
   T.set[i]=S.set[i];  }
bool setType::SetEmpty(setType S)
 { for(int i=firstValue;i<=lastValue;i++)
    if (S.set[i]==true) return false;}
bool setType::SubSetOf(setType S,setType T)
 {for(int i=firstValue;i<=lastValue;i++)
    if (S.set[i] && !T.set[i])  return false; }
void setType::Union(setType S,setType R, setType& T)
 {for(int i=firstValue;i<=lastValue;i++)
   T.set[i]=R.set[i]|| S.set[i];   }
void setType::AddElement(setType& S,int U) {S.set[U]=true;   }
```

```
void setType::SetDisplay(setType S)
 {for(int i=firstValue;i<=lastValue;i++) cout<<S.set[i]<<» «;  }
main()
{setType  LowerVowels,UpperVowels,Vowels, InLineVowels;
 char ch;
//Create a Set to represent uppercase vowels
UpperVowels.SetCreate(UpperVowels);
UpperVowels.AddElement(UpperVowels,>A>);
UpperVowels.AddElement(UpperVowels,>E>);
UpperVowels.AddElement(UpperVowels,>I>);
UpperVowels.AddElement(UpperVowels,>O>);
UpperVowels.AddElement(UpperVowels,>U>);
//Create a set to represent lowercase vowels
LowerVowels.SetCreate(LowerVowels);
LowerVowels.AddElement(LowerVowels,>a>);
LowerVowels.AddElement(LowerVowels,>e>);
LowerVowels.AddElement(LowerVowels,>i>);
LowerVowels.AddElement(LowerVowels,>o>);
LowerVowels.AddElement(LowerVowels,>u>);
//Create a set of uppercase and lowercase vowels
Vowels.Union(LowerVowels,UpperVowels,Vowels);
Vowels.SetDisplay(Vowels);
cout<<endl<<endl;
//Create a set InLineVowels which will be used to store any   vowel that
we find in the  //line of text
InLineVowels. SetCreate(InLineVowels);
 while (ch!=>0>)
   {cin>>ch;
    if (InLineVowels.IsElementOf(Vowels,ch))
      InLineVowels.AddElement(InLineVowels,ch); }
//Display the Set InLineVowels that appeared in the line
 InLineVowels.SetDisplay(InLineVowels);
 }
```

من المهم جداً، عزيزي الدارس، أن تلاحظ أننا استخدمنا تركيبة البيانات التجريدية setType من خلال العمليات المعرفة عليها فقط. فلإضافة 'A' إلى UpperVowels استخدمنا العملية AddElement ولم نحاول أبداً استغلال معرفتنا بأن UpplerVowels هي في حقيقة الأمر مصفوفة بولينية ذات بعد واحد. والفائدة المتوخاة من ذلك هي أنه فيما لو اضطررنا في المستقبل إلى تغيير طريقة تمثيل setType لتصبح قائمة متصلة linked-list على سبيل المثال بدلاً من مصفوفة بولينية، فإننا لن نضطر إلى تغيير برنامجنا أعلاه.

ومن المهم أيضاً، عزيزي الدارس، أن تلاحظ أهمية الالتزام، عند كتابة البرامج الفرعية المنفذة للعمليات المعرفة على setType، بترويسة تلك الدوال التي حددت في مرحلة التعريف بـ setType. وذلك حتى نستطيع استخدام هذه العمليات بدون الحاجة إلى دراسة (أو حتى رؤية) البرامج الفرعية المنفذة لتلك العمليات. ومن المهم أيضاً الالتزام بهذا الأمر عند تغيير طريقة تمثيل وتنفيذ التراكيب التجريدية حتى تبقى طريقة استدعاء هذه العمليات كما كانت.

6. الخلاصة

1. إن تراكيب البيانات التجريدية هي تلك التراكيب التي نستخدمها من خلال عمليات مُعّرفة مسبقاً عليها بدون الحاجة إلى معرفة تفاصيل تمثيل وتنفيذ تلك التراكيب والعمليات. والفوائد المتوخاة من ذلك هي:
2. فيما لو اضطررنا إلى تغيير طريقة تمثيل وتنفيذ التراكيب التجريدية لن نضطر إلى تغيير البرامج التطبيقية التي نستخدمها.
3. تصبح عملية كتابة البرامج التطبيقية أيسر وتتطلب جهداً أقل ووقتاً أقصر، وذلك لأننا نستطيع التركيز على هذه البرامج بدون الاهتمام بتفاصيل تمثيل التراكيب التجريدية التي تستخدم.
4. القدرة على الاستخدام المتكرر للبرامج الفرعية (Code Reuse) في برامج تطبيقية مختلفة.
5. تسهل التراكيب التجريدية عمل فريق البرمجة على البرامج التطبيقية الكبيرة، وذلك لأنها تجعل هذه البرامج أكثر توافقاً، إذ أنها تستخدم نفس النوع من التراكيب البيانية ومن خلال نفس العمليات.

7. لمحة عن الوحدة الدراسية الرابعة

في الوحدة التالية، عزيزي الدارس، سنناقش موضوع القوائم المتصلة بأنواعها المختلفة وسنعرض مقارنة بينها وبين المصفوفات ذات البعد الواحد.

8. إجابات التدريبات

تدريب (1)

```
bool setType::SetEqual(setType S, setType T)
  { for(int i=firstValue;i<=lastValue;i++)
      if (S.set[i]!=T.set[i]) return false;
  }
```

تدريب (2)

```
void setType::Intersection(setType S,setType R, setType& T)
  { for(int i=firstValue;i<=lastValue;i++)
     T.set[i]=R.set[i]&& S.set[i];
   }
```

تدريب (3)

```
void setType::Difference(setType S,setType R, setType& T)
  { for(int i=firstValue;i<=lastValue;i++)
     T.set[i]=R.set[i]&& !S.set[i];
  }
```

تدريب (4)

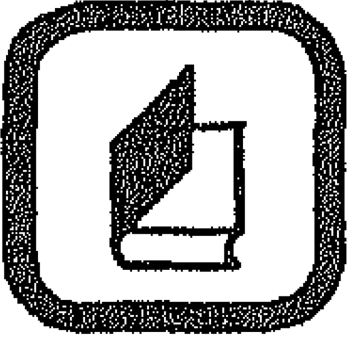
```
void setType::AddElement(setType& S,int U)
 {S.set[U]=true;
 }
```

تدريب (5)

```
void setType::RemoveElement(setType& S,int U)
  {S.set[U]=false;
  }
```

9. مسرد المصطلحات

- **إخفاء المعلومات التفصيلية Information Hiding**: المقدرة على تصميم وإنشاء قائمة ما وعند استخدامها لا ضرورة لمعرفة كيفية وتفاصيل تمثيلها.
- **أنواع التراكيب التجريدية Abstract Data Types**: تراكيب البيانات التي تستطيع استخدامها من خلال عمليات معينة معرفة على شكل دوال على هذه التراكيب، من دون الحاجة إلى معرفة تفاصيل تمثيل هذه التراكيب.
- **ترويسة الدالة Function Heading**: جزء من الدالة (أول سطر في الدالة) من أجزاء الدالة يخصص لوصف آلية التعامل مع هذه الدالة وبالتالي وصف للعمليات التي نرغب بتنفيذها على تركيبة البيانات.
- **المجموعات Sets**: المجموعة هي عدد من الأشياء غير المكررة المأخوذة من مدى (Universe) محدد من الأشياء. ولا يشترط أن تكون عناصر المجموعة مرتبة بأي ترتيب معين.



11. المراجع

1. Tremplay, J.P.; and Sorenson, P. G.; An Introduction to Data Structures with Applications, 2nd Edition, McGraw-Hill, 1984.
2. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice-Hall. 2000.
3. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, , Addison- Wesley, 1999.

الوحدة
الرابعة

القوائم Lists

محتويات الوحدة

1. المقدمة

1.1 تمهيد

أهلاً بك، عزيزي الدارس، إلى الوحدة الرابعة من كتاب "تركيب البيانات وتصميم الخوارزميات" وهي بعنوان "القوائم".

تعتبر القوائم من أكثر تراكيب البيانات أهمية وذلك لاستخداماتها المتعددة في مختلف المجالات. فكثيراً ما تتعامل برامجنا التطبيقية مع قوائم من الطلبة أو الموظفين أو البضائع وما إلى ذلك. وعليه فإنه من الأهمية بمكان، عزيزي الدارس، أن تعرف كيف تتعامل مع القوائم وكيف تمثلها وتنفذها بالطرق المختلفة وأن تعرف مميزات ومساوئ هذه الطرق. ومن أشهر هذه الطرق المصفوفات ذات البعد الواحد والقوائم المتصلة.

ومن الأخطاء الشائعة الاعتقاد بأن المقصود بالقائمة هي المصفوفة ذات البعد الواحد. إن إحدى طرق تمثيل القوائم هي باستخدام المصفوفات وما هي إلا طريقة واحدة لها مميزاتها المهمة ولها أيضاً مساوئها التي قد لا تجعلها الطريقة الأفضل في كثير من الأحيان.

ومن المهم أيضاً أن نتعامل مع القائمة على أنها تركيبة بيانات تجريدية لها تعريفها المنطقي ولها عملياتها التي نستطيع تنفيذها عليها وذلك للأسباب المتعددة التي ناقشناها في الوحدة السابقة.

وكثيراً ما نحتاج إلى التعامل مع القوائم المرتبة بطريقة معينة. كأن ترتب قائمة من الطلبة حسب الترتيب الأبجدي لأسمائهم أو حسب معدلاتهم. وذلك لكون القوائم المرتبة أسهل للتعامل فهي أفضل للعرض وللبحث عن قيمة معينة. فعند البحث عن قيمة معينة في قائمة مرتبة لا تحتوي على تلك القيمة لا نحتاج إلى البحث في كل القائمة، إذ نستطيع التوقف عندما نصل إلى قيمة في القائمة أكبر (أو تلي) القيمة التي نبحث عنها حسب الترتيب المتبع. وعليه تكون عملية البحث أكثر كفاءة من حيث الوقت اللازم لإجرائها. ومن الناحية الأخرى قد يؤدي استخدامنا للقوائم المرتبة إلى تعقيد عمليتا الإضافة والحذف من القائمة قليلاً. إذ يجب أن تتم الإضافة (وكذلك الحذف) في الموقع الصحيح في القائمة. وسنفترض في بقية هذه الوحدة أننا نتعامل مع قوائم مرتبة وذلك لكونها الأكثر استخداماً ولكون تنفيذ العمليات عليها أكثر تعقيداً.

2.1 أهداف الوحدة

ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1. توضح المفاهيم المختلفة المتعلقة بالقوائم كتراكيب بيانات تجريدية.
2. تمثل القوائم وتنفذ العمليات الخاصة بها بطرق مختلفة.
3. تستخدم القوائم المتصلة.
4. تحاكي القوائم المتصلة باستخدام المصفوفات.
5. تميز بين الأنواع المختلفة للقوائم المتصلة.
6. تمثل المصفوفات الثنائية والمصفوفات متعددة الأبعاد.
7. تميز المصفوفات الشتيتة وتمثلها بطرق مناسبة.

3.1 أقسام الوحدة

تقسم هذه الوحدة إلى خمسة أقسام رئيسة، ترتبط بقائمة الأهداف السابقة.

حيث نناقش في القسم الأول القائمة كتركيبة بيانات تجريدية أي بصفاتها العامة المجردة، وهذا يرتبط بالهدف الأول. ويوضح القسم الثاني كيفية تمثيل القوائم وكيفية تمثيل العمليات الخاصة بها وهذا القسم مرتبط بالهدفين الثاني والثالث. ويبين القسم الثالث كيفية محاكاة القوائم المتصلة باستخدام المصفوفات ويرتبط بالهدف الرابع. ويناقش القسم الرابع الأنواع المختلفة للقوائم المتصلة وهو مرتبط بالهدف الخامس. أما القسم الخامس وهو الأخير فيناقش المصفوفات الثنائية ومتعددة الأبعاد وطرق تمثيلها في ذاكرة الحاسوب، ويناقش أيضاً نوعاً خاصاً من المصفوفات الثنائية تعرف باسم المصفوفات الشتيتة وطرق تمثيلها، وهو مرتبط بالهدفين السادس والسابع.

4.1 القراءات المساعدة

لقد جاءت هذه الوحدة شاملة لكل المفاهيم الأساسية الخاصة بالقوائم، ومن المفيد أيضاً أن تعود، عزيزي الدارس، إلى المصادر التالية وبخاصة في الحالات التي ترغب فيها بالحصول على أمثلة أخرى وعلى طرق أخرى لتمثيل التراكيب البيانية التي تم عرضها في هذه الوحدة. فهناك أساليب أخرى للتمثيل وإجراء العمليات لم نتعرض لها بالتفصيل في هذه الوحدة وبخاصة ما يتعلق منها باستخدام أسلوب المصفوفات المتوازية والمصفوفات المركبة التي أشرنا إليها خلال هذه الوحدة. ومن المصادر الهامة في هذا الصدد:

1. Lewis, T.G., and Smith, M. Z., Applying Data Structures.: Atlanta: Houghton Mifflin, 1976, pp. 23 – 43.
2. Lipschutz, Seymour; Schaum, s., Outline of Theory and Problems of Data Structures. New York: McGraw-Hill, 1986, pp. 67-113 & 114-163.
3. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice- Hall. 2000.
4. Malik, D.S. Data Structures Using C++, 1st Edition, Course Technology, Inc., 2003.
5. Main, Michael Data Structures & Other Objects Using C,++, 3d Edition, Addison-Wesley, 2004.

2. القائمة كتركيبة بيانات تجريدية

نظراً للفوائد الهامة للتراكيب التجريدية (انظر الوحدة الثالثة من المقرر) فإننا سنتعامل مع القوائم كتراكيب بيانية تجريدية. وعليه وقبل التفكير بكيفية تمثيل القوائم وتنفيذها علينا تعريف القوائم والعمليات التي تستخدم من خلالها.

القائمة المرتبة: هي مجموعة من الأشياء مرتبة بشكل خطي حسب علاقة ترتيب بين عناصرها.

إن العمليات التي تستخدم القائمة من خلالها هي:

1. إنشاء القائمة (Listcreate)

وتنفذ هذه العملية كدالة تقوم بإنشاء قائمة فارغة

```
void Listcreate(slist* L);
```

2. الإضافة (insertion)

يمكن الإشارة إلى عملية الإضافة بصفة مجردة بالصيغة التالية:

```
slist* insert(slist* list,int element);
```

وهذه الصيغة تعني إضافة العنصر element في القائمة list. أي أننا بحاجة إلى معلومتين حتى نقوم بهذه العملية، وهما: العنصر الذي نرغب في إضافته، واسم القائمة التي نرغب في إضافة العنصر إليها. فإذا علمنا أن القائمة هي: $(a_1 , a_2 ,..., a_n)$، فإن إضافة العنصر x إلى الموضع p فيها سيؤدي إلى تعديل القائمة لتصبح على النحو التالي:

$(a_1 , a_2 , ..., ap_{-1} , X , a_{p+1} , ..., a_n)$

3. الحذف (Deletion)

أما فيما يتصل بالحذف، فيمكن الإشارة إليه بالصيغة المجردة التالية:

```
slist* deletee(slist* L, int element);
```

أي حذف العنصر element من القائمة L. ومعنى ذلك أننا نحتاج إلى معلومتين حتى نقوم بهذه العملية، وهما: العنصر الذي نرغب في حذفه، وإسم القائمة التي سيتم الحذف منها. وبناء على ذلك، إذا علمنا أن القائمة هي: (a1 , a2 ,..., an) فإن حذف العنصر الواقع في الموضع p سيؤدي إلى تعديل وضع القائمة لتصبح على النحو التالي:

$(a_1 , a_2 , ..., a_p-1 , a_p+1 , ..., a_n)$

4. الاستقصاء وتحديد الموقع (searching and finding location)

إن عملية الاستقصاء وتحديد موقع أحد العناصر ذات أهمية خاصة بالنسبة لعمليتي الإضافة والحذف، كما لاحظنا أعلاه. وهي ذات أهمية مماثلة بالنسبة لعملية الاسترجاع

وعملية تحديث العناصر. والصيغة العامة الأساسية لعملية الاستقصاء يمكن أن تتخذ الشكل التالي:

slist locateP(slist* list, slist* loc,int x);*

أي حدد موقع العنصر x في القائمة list. ومعنى ذلك أننا بحاجة إلى معلومتين أساسيتين حتى نقوم بهذه العملية، وهما: العنصر الذي نرغب في البحث عنه، واسم القائمة التي نرغب في إجراء عملية الاستقصاء فيها. ونتيجة هذه العملية هي قيمة تحدد موضع العنصر loc داخل القائمة أو الإشارة إلى أنه غير موجود. وفي حالة تكرار العنصر في أكثر من موضع، فإن النتيجة المعطاة تتصل بالموضع الذي ورد فيه العنصر للمرة الأولى.

5. تحديد العنصر السابق والعنصر اللاحق(Determine the predecessor and (the Successor

وبالإضافة إلى ما ذكر فإن هناك بعض العمليات الأخرى التي تجري على هامش عملية الاستقصاء، وهي أيضاً ذات أهمية كبيرة في عمليات الحذف والإضافة بشكل خاص. ومن أهم هذه العمليات تحديد السابق واللاحق، وتحديد العنصر الأول وتحديد العنصر الأخير. ونشير هنا فقط إلى عملية تحديد السابق واللاحق. ويمكن التعبير عنها بالصيغة التالية:

slist find(slist* pred, slist* succ, int x, slist* list);*

أي حدد موقع pred وموقع succ بالنسبة للقيمة x في القائمة list . وكما هو واضح، فإن العنصر الأول لا سابق له وبذلك تكون نتيجة البحث عن السابق الثابت الخاص NULL وقيمته تعتمد على طريقة تنفيذ القائمة فهو NULL إذا استخدمنا المؤشرات وهي صفر إذا استخدمت المصفوفات. وبالمثل فإن العنصر الأخير لا تالي له، وبذلك نخلص إلى نتيجة مماثلة وهي أن قيمة succ هي NULL.

6. الاسترجاع (Retrieval)

الاسترجاع هو خطوة تالية لعملية الاستقصاء وتحديد الموقع، ويمكن التعبير عنها مجردة كما يلي:

slist retrieve(slist* list, int p, int& element);*

أي استرجاع العنصر الذي يحتل الموضع p في القائمة list. ومعنى ذلك أننا بحاجة إلى معلومتين للقيام بهذه العملية، وهما: موقع العنصر الذي نرغب في استرجاعه، واسم القائمة التي نرغب في استرجاع العنصر منها. وعلى خلاف عمليتي الإضافة والحذف فإن عملية الاسترجاع لا تعدل في وضع القائمة التي تتم عليها عملية الاسترجاع. وإذا كانت

قيمة الموضع صفراً أو NULL فإن عملية الاسترجاع ستؤدي إلى الوصول إلى النتيجة نفسها (أي NULL).

7. تعديل قيمة عنصر (Update)

تغيير قيمة أحد العناصر أو استبداله بقيمة أخرى، وهي تتخذ الصيغة التالية:

```
slist* replace(int x1,int x2, int  p, slist* list);
```

أي بمعنى استبدال العنصر x1 الوارد في الموضع p في القائمة list بالعنصر الجديد أو القيمة الجديدة x2

8. تحطيم القائمة وإزالة عناصرها (Destroy List)

عملية إزالة جميع العناصر وتحويل القائمة إلى قائمة خالية، ويمكن أن تتخذ الصيغة التالية:

```
slist* makeNull(slist* list);
```

9. طباعة القائمة (Display List)

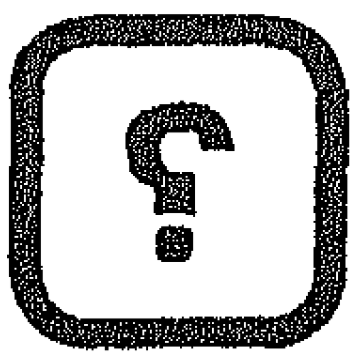
عملية طباعة جميع عناصر القائمة، وهي تتخذ الصيغة التالية:

```
void printList(slist* list);
```

10. استعراض قائمة (Traversal)

وتعني المرور على كل عنصر فيها وتنفيذ خطوات معينة على كل منهم كطباعة قيمهم أو تعديلها أو إحصائها وما إلى ذلك. وتأخذ الصيغة:

```
void traversal(slist* list);
```



أسئلة التقويم الذاتي (1)

أجب بنعم أو لا:

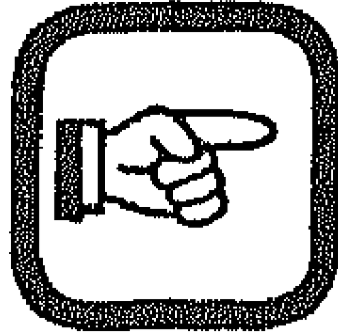
1. جميع عناصر القائمة الممثلة بطريقة خطية متصلة ينبغي أن تكون من نمط (نوع) واحد.
2. المعلومات التي نحتاجها لإجراء عملية الإضافة هي: القيمة المراد إضافتها، ومكان الإضافة، واسم القائمة.
3. إن عملية تحديد موقع العنصر داخل القائمة مسألة ضرورية في حالات الإضافة والحذف والاسترجاع والتغيير.

3. تمثيل القوائم وتنفيذ عملياتها

في هذا القسم، عزيزي الدارس، سنقوم بمناقشة طريقتين رئيستين لتمثيل القوائم، الأولى باستخدام القوائم المتصلة والثانية باستخدام المصفوفات. وسنقوم بتنفيذ بعض العمليات على القوائم باستخدام كلا الطريقتين.

1.3 تمثيل القوائم باستخدام القوائم المتصلة

لقد أشرنا فيما سبق إلى أن من الممكن تمثيل القائمة على هيئة سلسلة من العناصر، كل منها يشير إلى الآخر بواسطة رابطة خاصة نطلق عليها، بصفة عامة، اسم المؤشر (pointer)، وهو يحتوي على قيمة معينة تحدد موقع تخزين العنصر التالي في القائمة. وللمزيد من التوضيح، دعنا نتأمل المثال التالي:



مثال (1)

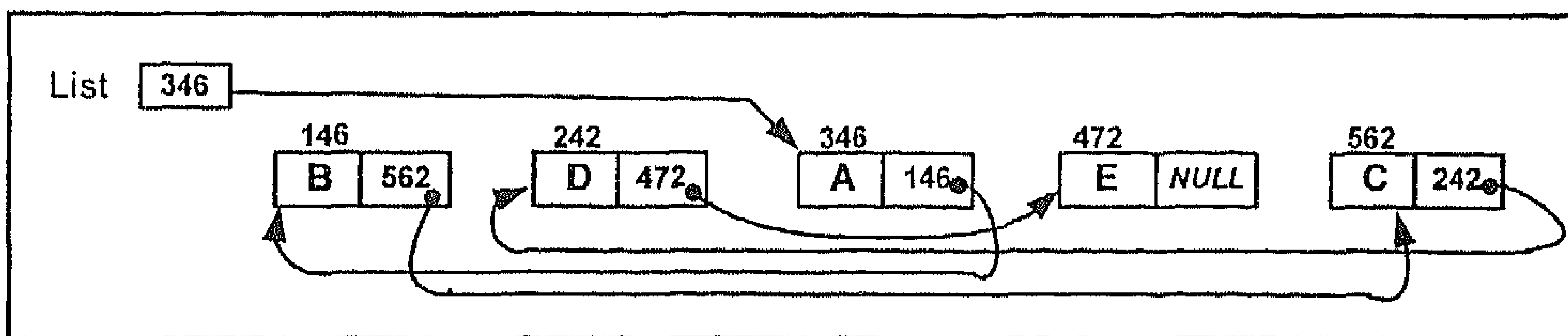
افترض أن لدينا خمس قيم رمزية، هي: (A, B, C, D, E) وأن كلاً منها مخزن في مكان مستقل في الذاكرة على النحو المبين في الشكل (1) أدناه. وكما هو واضح من هذا الشكل، فإن القيمة A مخزنة في الموضع المشار إليه بالعنوان المطلق (346)، والقيمة B مخزنة في موضع بعيد عنها وهو الموضع المعنون بالرقم (146)، والقيمة C في مكان آخر بعيد... وهكذا. وباختصار، لا يوجد بين هذه القيم في الذاكرة أي تجاور ولا أي رابطة تجمعها معاً في سياق منتظم، كما يظهر في الشكل (2 - أ).

562	472	346	242	146
C	E	A	D	B

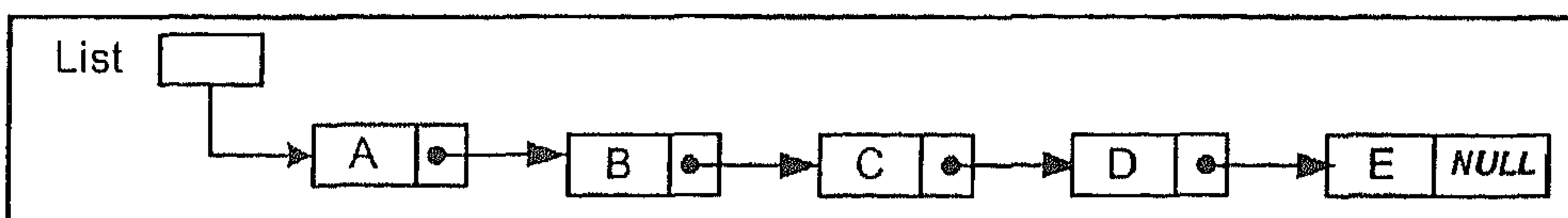
الشكل (1): مجموعة من القيم مخزنة في أماكن متفرقة من الذاكرة

افترض الآن أننا نريد أن نشكل من هذه القيم قائمة منتظمة، تتسلسل فيها القيم بطريقة هجائية، ونشير إليها باسم واحد. فمعنى ذلك أننا الآن بحاجة إلى أسلوب معين للربط بين هذه القيم والإشارة إليها جميعاً من خلال متغير واحد، وهو list على سبيل المثال. فالخطوة الأولى، إذن، هي تخصيص مكان معين في الذاكرة ليشير إلى العنصر الأول في هذه القائمة. وتتم الإشارة من خلال استخدام العنوان المطلق لهذا العنصر وكل

عنصر تال. والخطوة التالية هي أن نقوم بالربط بين العنصر والعنصر الذي يليه من خلال مؤشر يصاحب كل عنصر من هذه العناصر، وبذلك ستكون النتيجة على النحو الموضح في الشكل (2 - ب).



أ- السياق الفيزيائي لقائمة متصلة في الذاكرة



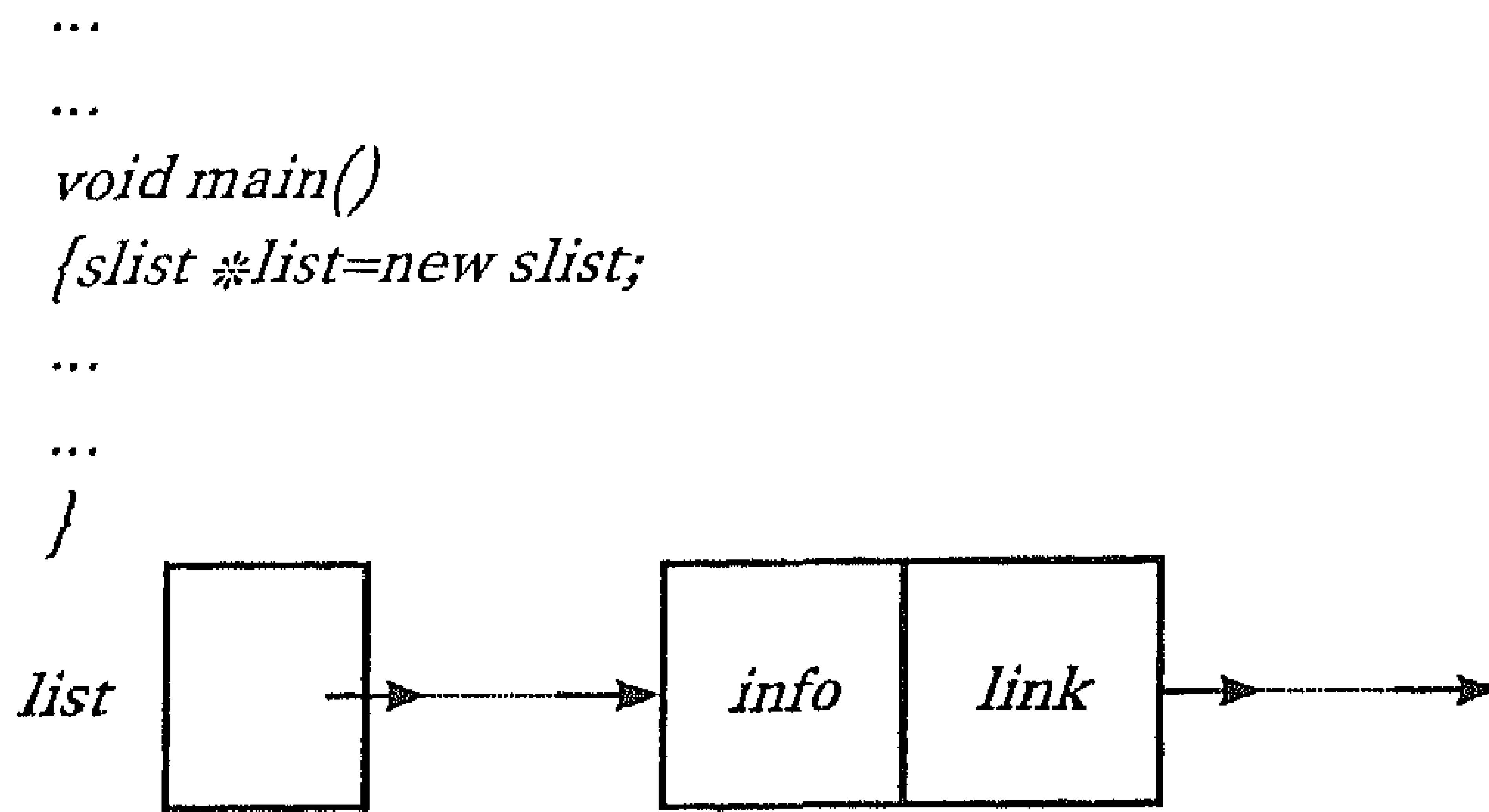
ب- السياق المنطقي للقائمة المتصلة LIST

الشكل (2): التصوران الفيزيائي والمنطقي لقائمة متصلة

وبناء على ذلك، أصبح كل عنصر يشير إلى الآخر، وشكلنا بذلك قائمة تتصل عناصرها معاً مكونة سياقاً خطياً منتظماً. ومن هنا، يصطلح عليها باسم القوائم المتصلة. وكما تلاحظ، من الشكل (2)، فإن كل عنصر من العناصر الخمسة لهذه القائمة يتكون من جزئين: الأول يمثل القيمة المعنية في القائمة (مثل A وB .. الخ)، والآخر يضم عنوان الموضع الذي يتبع هذه القيمة في القائمة وهو ما نسميه المؤشر.

وبناء على ما تقدم نستطيع تمثيل القائمة slist في لغة سي++ كقائمة متصلة كما هو مبين في الشكل (3):

```
class slist
{private:
    slist* link;
    char info;
public:
    slist* insert(slist*,int element);
     ...
     ...
 };
```



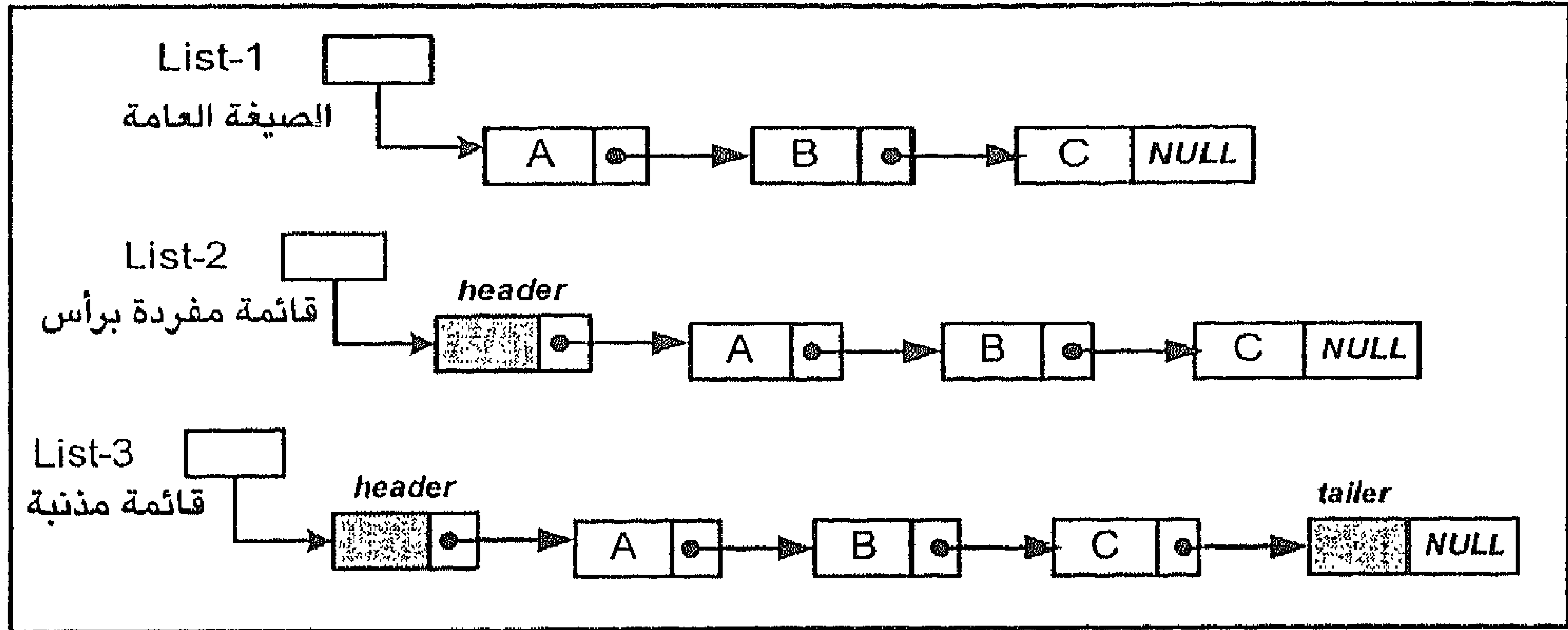
الشكل (3): صيغة تكوين العنصر الواحد في القائمة المتصلة

وفي هذه الحالة فإن العناصر من نوع char لأننا نخزن رموزاً في القائمة.

وينبغي أن نلاحظ أن المؤشر الخارجي list يشير إلى العنصر الأول في القائمة، وأن العنصر الأخير لا يشير إلى أي عنصر آخر، ومثلنا هذه الحقيقة في الشكل (2) بالقيمة NULL في الحقل الثاني (link). وهذه القيمة ذات أهمية خاصة في العمليات التي تُجرى على القوائم المتصلة البسيطة، لأنها تخبرنا عن نهاية القائمة.

2.3 تنفيذ عمليات القوائم باستخدام القوائم المتصلة المفردة

إن القوائم المتصلة السابقة هي قوائم مفردة ونعني بذلك أبسط أنماط القوائم المتصلة وأكثرها شيوعاً. وهي تتخذ شكلاً خطياً وتسير في اتجاه واحد وقيمة آخر مؤشر فيها هي NULL. ولعلك تلاحظ أن جميع التراكيب المتصلة التي ورد ذكرها حتى الآن تدخل ضمن هذا النوع من القوائم المتصلة. وبالإضافة إلى الصيغة العامة التي ترد فيها القوائم المفردة، فإنها قد ترد بصيغتين أخريين: إحداهما تمتاز بوجود عنصر إضافي خاص في بدايتها نشير إليه باسم المقدمة أو الرأس (header) والصيغة الأخرى تتميز بوجود عنصر إضافي خاص في نهايتها بالإضافة إلى الرأس، نشير إليه باسم الذيل أو الذنب (tailer). والهدف منهما هو تخزين بعض المعلومات الخاصة عن القائمة كعدد العناصر، أو عدد مرات التعديل التي جرت على القائمة أو غير ذلك من المعلومات التي يرتئي المبرمج تضمينها في هذه العناصر الإضافية الخاصة وقد يتركها بلا شيء. والشكل (4) يعطي الصيغ الثلاثة المختلفة التي ترد بها القوائم المفردة. وسنركز فيما يلي من عمليات وخوارزميات على الصيغة العامة الواردة أولاً في هذا الشكل.



الشكل (4): الصيغ المختلفة التي ترد عليها القائمة المفردة

1.2.3 إنشاء القائمة List Create

كل ما تصنعه هذه الدالة هو إنشاء قائمة فارغة أي NULL.

```
void listCreate(slist* L)
  {L=NULL;}
```

ولإنشاء قائمة غير فارغة (تحتوي على عدد من العناصر) فإن الدالة insert تستخدم العدد المطلوب من المرات كل مرة لإضافة عنصر جديد وذلك بعد استخدام الأمر listCreate لإنشاء قائمة فارغة.

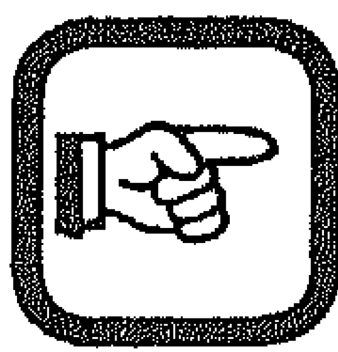
2.2.3 استعراض القوائم المتصلة (list traversal)

أما وقد قمنا ببناء القائمة المتصلة فإن بإمكاننا أن نجري عليها بعض العمليات الأساسية. ومن بين هذه العمليات عملية استعراض عناصر القائمة، بحيث نقوم بزيارة كل عنصر مرة واحدة، وإجراء أية معالجة مطلوبة على هذا العنصر. والخوارزمية التالية تحدد تفاصيل القيام بهذه العملية:

الخوارزمية (1):

```
void slist::traversal(slist* list)
    {slist* current;
     current=list;
     while(current!= NULL)
      {       process(current-> info);
              current= current->link;
      }
    }
```

وكما تلاحظ، عزيزي الدارس، فإن هذه الخوارزمية تستخدم مؤشراً متحركاً هو current لزيارة العناصر التي تتكون منها القائمة. فهو يبدأ بالعنصر الأول من خلال تنفيذ الجملة current=list;. فإذا كانت قيمة list هي NULL فإن معنى ذلك أن القائمة خالية وبذلك ينتقل التنفيذ إلى نهاية الخوارزمية، وإلا فإنه سينتقل إلى تنفيذ جملة process...، حيث تتخذ هذه الجملة شكل استدعاء لخوارزمية أخرى اسمها process للقيام بنوع من المعالجة على العنصر الذي يشير إليه المؤشر الخارجي current ثم يتحرك خطوة إلى الأمام لمعالجة العنصر التالي من خلال الجملة current=current->link;. ويستمر هكذا حتى تصبح القيمة (current == NULL) حينئذ يخرج من التركيب الدوراني ويتوقف التنفيذ ولتوضيح فكرة هذه الخوارزمية، دعنا نتأمل المثال التالي:



مثال (2)

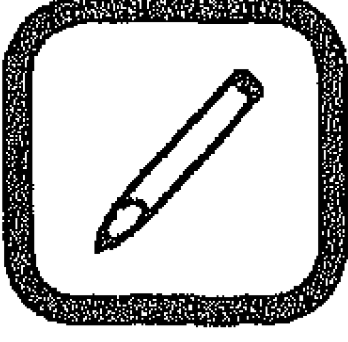
افترض أن لدينا قائمة متصلة ونرمز إليها بالاسم list، ونرغب أولاً في طباعة عناصرها واحداً تلو الآخر، ونرغب أيضاً بالقيام بإجراء مستقل لحساب عدد العناصر التي تتضمنها هذه القائمة. يمكن، لتحقيق ذلك، أن نستخدم الدالتين "printl" و"count" كما هو مبين أدناه:

```
void slist::printl(slist* list)
    {slist* current;
  current=list;
  while (current != NULL)
   {cout<<current->info;
    current=current->link;
   }
  }
void slist::count(slist* list, int& num)
    {slist* current;
  num=0;
  current=list;
  while (current != NULL)
   {num++;
    current=current->link;
   }
  }
```

وكما تلاحظ، فإنه ليس هناك أي فرق أساسي بين الدالتين المعرفتين في هذا المثال سوى الجزء المعبر عن عملية process المذكور في الخوارزمية (1). فقد عبرنا عن:

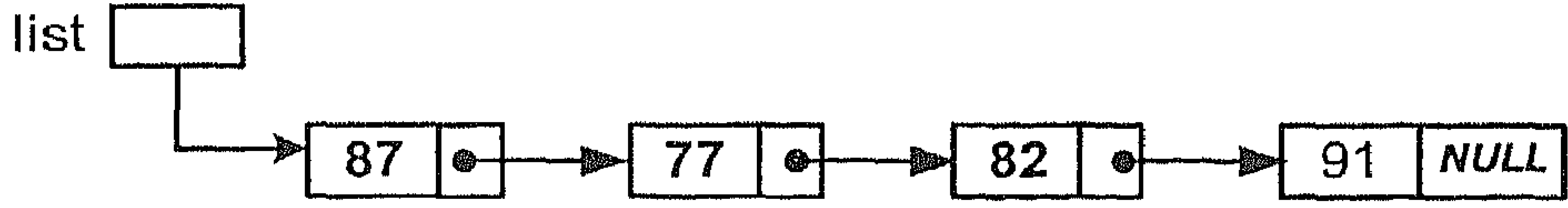
process(current-> info); كما وردت في الخوارزمية بالجملتين التاليتين في المثال:

cout<<current->info;
num++;



تدريب (1)

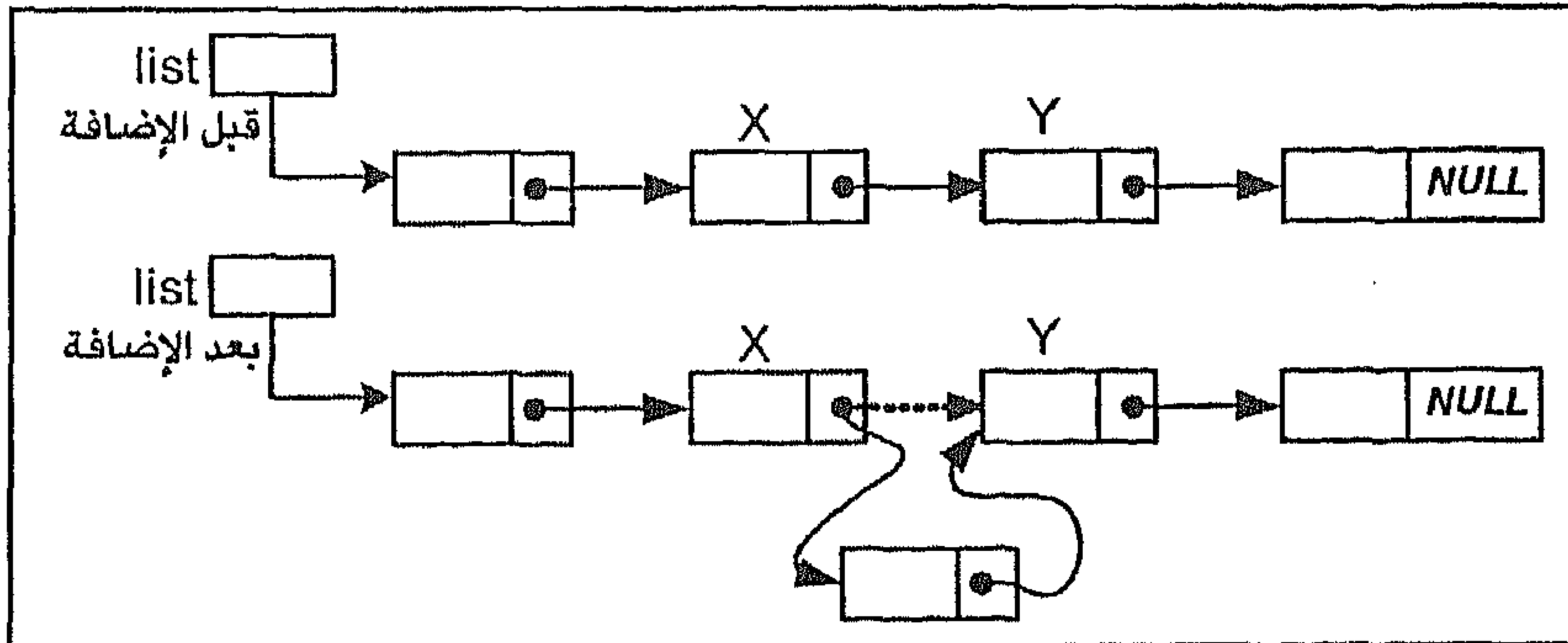
استخدم الرسم التالي (الشكل (5)) لكتابة دالة في لغة سي++ على غرار المثال السابق لمعرفة القيمة الدنيا. ثم قارن بين إجابتك والإجابة المعطاة في نهاية الوحدة، ولاحظ أية فروق.



الشكل (5): قائمة متصلة خاصة بالتدريب (1)

1. إضافة عناصر جديدة إلى القوائم المتصلة (Insertion)

تقتضي عملية إضافة عنصر جديد إلى القائمة تحديد المكان الذي ستتم فيه عملية الإضافة داخل القائمة، كما أشرنا في بداية هذه الوحدة. ولما كانت إضافة العنصر الجديد تتم في كثير من الأحيان بين عنصرين من عناصر القائمة، فإن هذه العملية تنطوي على إحداث تغيير في سلسلة المؤشرات. فإذا افترضنا أن العنصر الجديد سيضاف بين العنصرين X وY فإن ذلك يعني أننا سنفك الارتباط بين هذين العنصرين ونجعل X تشير إلى العنصر الجديد، ونجعل هذا الأخير يقوم بدوره بالإشارة إلى Y (كما هو واضح في الشكل (6) أدناه).



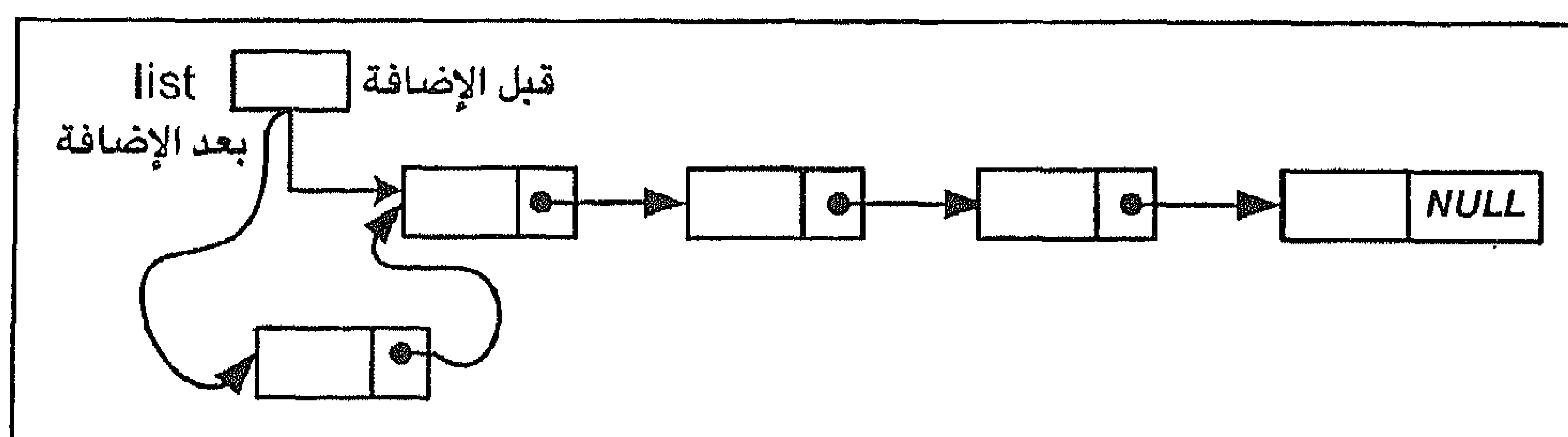
الشكل (6): إضافة عنصر جديد بين عنصرين آخرين في القائمة

والوضع الثاني الذي نصادفه هو إضافة العنصر الجديد إلى بداية القائمة. وفي هذه الحالة، فإن الأمر يستدعي فك الارتباط القائم بين المؤشر الخارجي list والقائمة، وجعله يشير إلى العنصر الجديد الذي يقوم هو الآخر أيضاً بدوره بالإشارة إلى العنصر الأول (السابق). وبذلك يصبح العنصر الجديد هو العنصر الأول في القائمة بعد عملية الإضافة، كما هو واضح في الشكل (7). وهناك في الحقيقة ثلاث حالات يمكن أن تحدث فيها الإضافة في بداية القائمة وهي:

الأولى: عندما نرغب، لأسباب فنية، في إجراء الإضافة في البداية كما هو الحال في المكدسات التي تستخدم التمثيل القائم على المؤشرات.

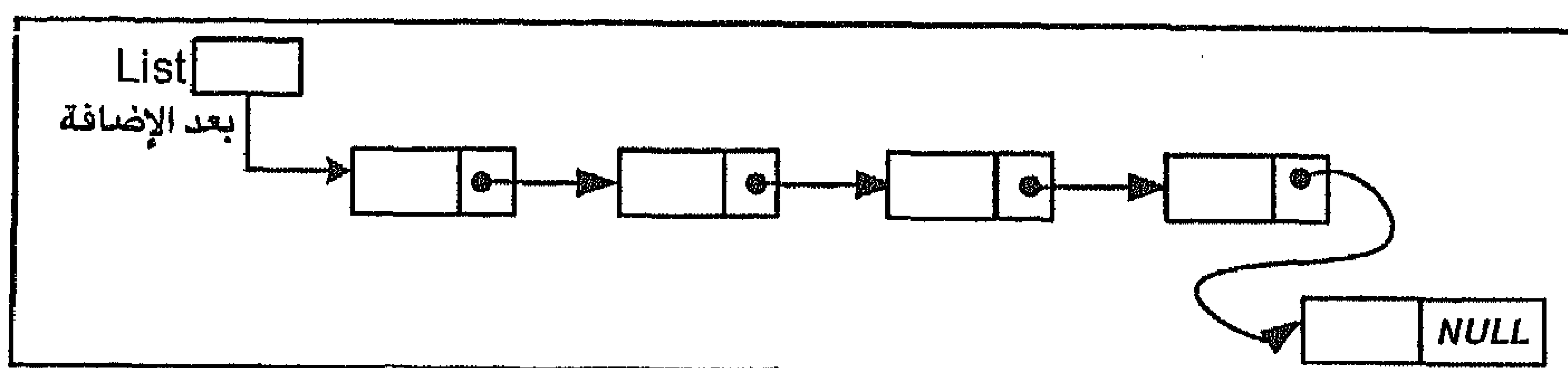
الثانية: عندما تكون القائمة مرتبة وفق نظام معين وجاء العنصر المضاف، بحكم ترتيب القيم، في بداية القائمة.

الثالثة: عندما تكون القائمة خالية، والعنصر المضاف في هذه الحالة هو العنصر الأول والوحيد في القائمة.



الشكل (7): إضافة عنصر جديد إلى بداية القائمة

أما الوضع الثالث فهو إضافة العنصر الجديد إلى آخر القائمة. وعلى خلاف الحالتين السابقتين، فإن الأمر لا يحتاج هنا إلى أي نوع من فك الارتباط بين عنصر وآخر أو بين المؤشر الخارجي والعنصر الأول. وكل ما يحتاجه الموقف هو تغيير قيمة مؤشر العنصر الأخير في القائمة من NULL إلى عنوان العنصر الجديد، وجعل قيمة مؤشر العنصر المضاف NULL وذلك على النحو الموضح في الشكل (8).



الشكل (8): إضافة عنصر جديد إلى نهاية القائمة

وقد تحدث الإضافة في نهاية القائمة في حالتين، الأولى: هي الطوابير القائمة على التمثيل المتصل، والثانية: هي القوائم المرتبة وفق نظام معين بحيث يكون وضع العنصر الجديد في آخر التسلسل القائم للقيم.

وفيما يلي نقوم بعرض الخوارزمية التي تأخذ بالحسبان هذه الأوضاع الثلاثة، مع الافتراض بأن القائمة مرتبة وفق نظام معين.

الخوارزمية (2):

```
slist * slist::insert(slist * list, int element)
    {slist *loc;
     slist * newNode = new slist;
     newNode->info = element;
     newNode->link =NULL;
     loc=list;
     loc=locateP(list,loc,element);
     if (loc==NULL)    //insert as first node
      {newNode->link = list;
       list= newNode;
      }  // end if
     else  // insert else where
      {newNode->link =loc->link;
       loc->link=newNode;
      } //end else
     return list;
    }  // end insert
```

وكما تلاحظ فإن هذه الخوارزمية تقوم باستدعاء خوارزمية أخرى لتحديد موضع الإضافة وهي locateP. وتفاصيل هذه الخوارزمية تعتمد إلى حد كبير على طريقة ترتيب القيم في القائمة المتصلة، وسنفترض أن القيم مرتبة ترتيباً تصاعدياً عند كتابة هذه الخوارزمية. وفيما يلي نقوم بتعريف هذه الخوارزمية في لغة C++.

الخوارزمية (3):

```
slist* slist::locateP(slist * list, slist * loc,int element)
    { bool found;
      slist *previous,*next;
      if (list==NULL || element< list->info)
               {loc=NULL;   //insertion in the beginning
       }
```

```
else   //search for location
 {previous=list;
  next=list->link;
  found=false;
  while (next !=NULL && !found)
    {if (element < list -> info )
      {loc=previous;
       found=true;
      }  //end if
    else  // update pointers
     {previous=next;
      next=next->link;
      } // end else
     }//end while
     loc=previous; // insertion may be at end
   }//end else
 return loc;
}// end locateP
```

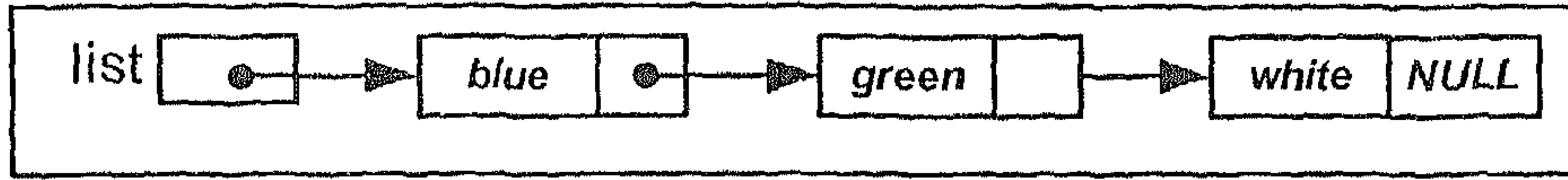
ولو أمعنت النظر، عزيزي الدارس، في خوارزمية الإضافة لوجدت أنها تنقسم إلى ثلاثة أجزاء: الأول يتضمن الجمل الثلاثة الأولى ومهمته هي إيجاد العنصر الجديد نفسه. والثاني يتضمن جملة استدعاء الخوارزمية الأخرى locateP، ومهمته هي إيجاد المكان المناسب لإضافة العنصر الجديد. وأما الثالث فيتضمن التركيب الشرطي if-else، ومهمته هي إجراء عملية الإضافة نفسها. أما بالنسبة للدالة locateP فإنها تتعامل مع الأوضاع الثلاثة التي سبق أن حددناها. فالجزء الأول للتركيب الشرطي يحدد ما إذا كانت الإضافة ستتم في بداية القائمة، والجزء الثاني منها، الذي يلي كلمة else مباشرة، يحدد المكان داخل القائمة نفسها. وفي حالة خروجنا من الدوران وكانت نتيجته أن قيمة next هي NULL فإن معنى ذلك أن مكان الإضافة هو نهاية القائمة، وهذا الوضع هو ما نتعامل معه في الجملة الأخيرة من الدالة locateP.



مثال (3)

لكي نوضح كيف تعمل خوارزمية الإضافة وخوارزمية إيجاد الموقع التابعة لها، دعنا نفترض بأن لدينا القائمة المبينة في الشكل (9 - أ) المرتبة هجائياً وأننا نرغب في إضافة

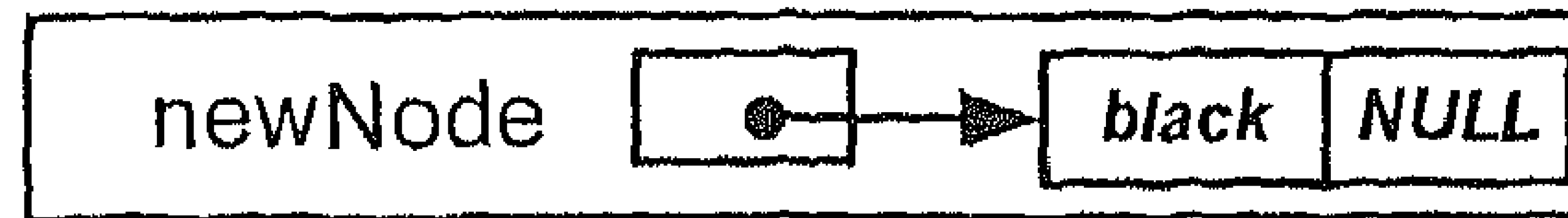
ثلاثة عناصر جديدة إليها، كل منها على انفراد، وهي: (black, orange, yellow) أي أن قيمة element في كل مرة، هي واحدة من القيم الثلاثة.



الشكل (9 - أ): القائمة قبل الإضافة

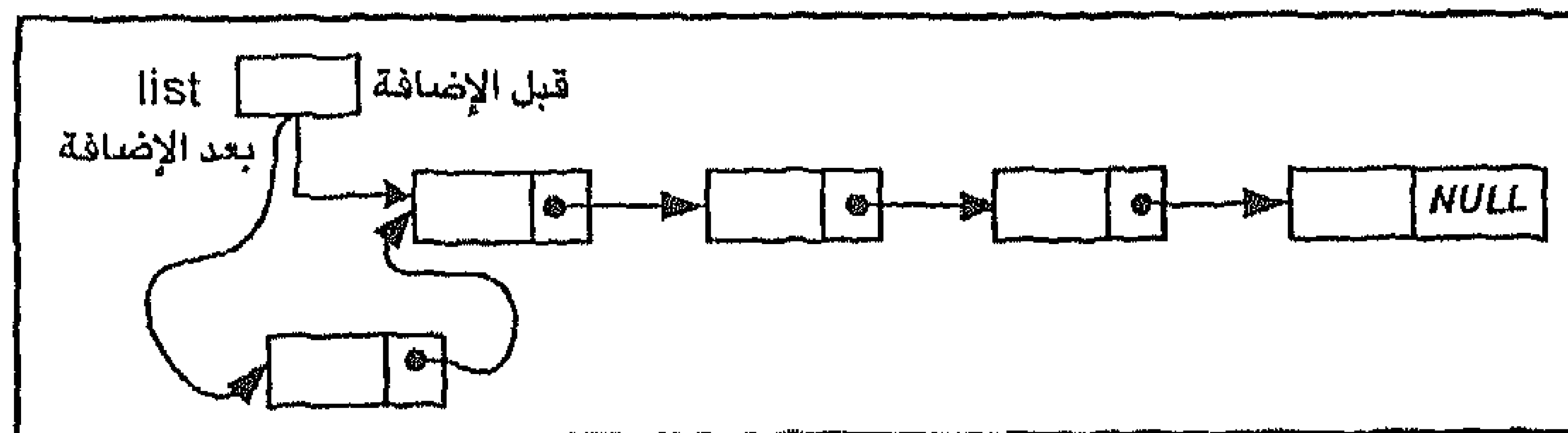
أولاً: إضافة العنصر 'black'

بعد أن يتم تنفيذ الجمل الثلاثة الأولى في الدالة insert ينتج لدينا الوضع التالي (الشكل (9 - ب))



الشكل (9 - ب): إيجاد العنصر الأول

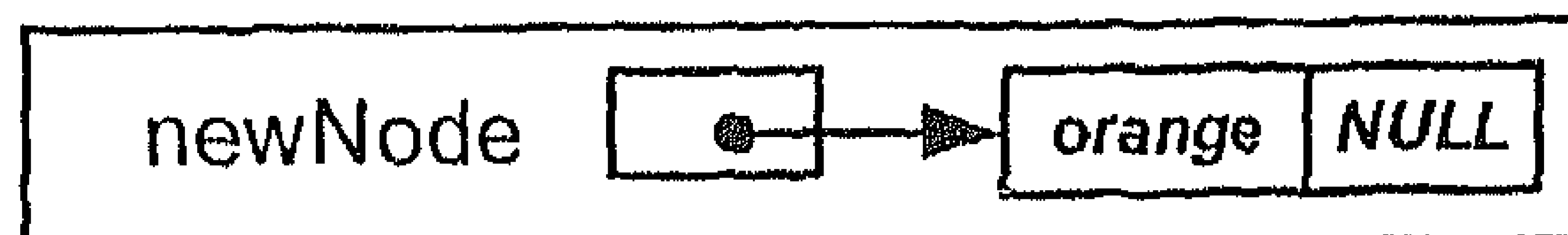
ثم يتلو ذلك استدعاء الدالة locateP حيث يتم اختبار الجزء الأول من الشرط وهو (list == NULL) وتكون النتيجة false ثم يتم اختبار الجزء الثاني وهو (element < list -> info) وتكون النتيجة true . وبذلك يتم إسناد القيمة NULL إلى المؤشر loc ونعود إلى الدالة insert، حيث يتم اختبار الشرط (loc == NULL) والنتيجة هي true. وبذلك تتم إضافة العنصر الجديد إلى بداية القائمة ويصبح الوضع كما هو في الشكل (9 - ج).



الشكل (9 - ج): القائمة بعد إضافة black إلى البداية

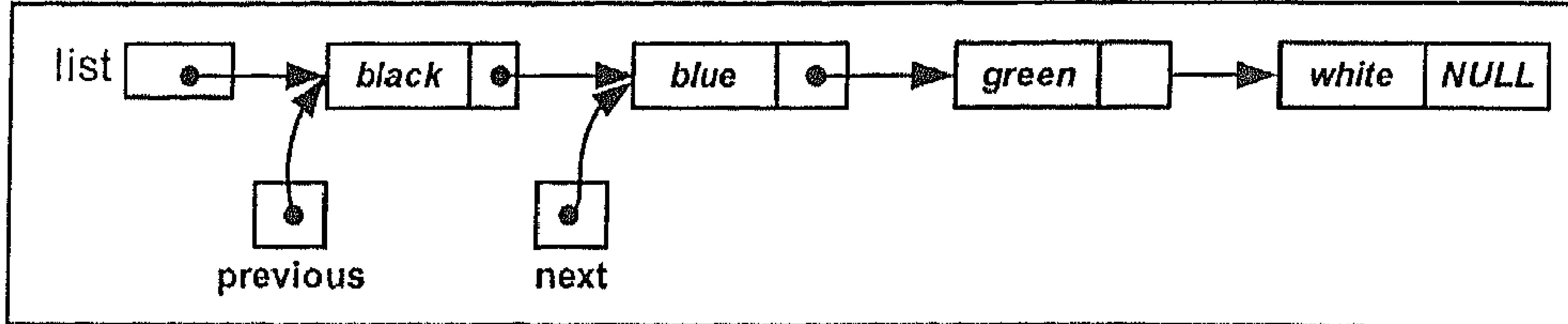
ثانياً: إضافة العنصر 'orange'

ثم نقوم مرة أخرى باستدعاء الدالة insert، وقيمة element في هذه المرة هي 'orange' وتكون النتيجة تنفيذ الجمل الأربعة الأولى لنحصل على الوضع الموضح في الشكل (9 - د) .



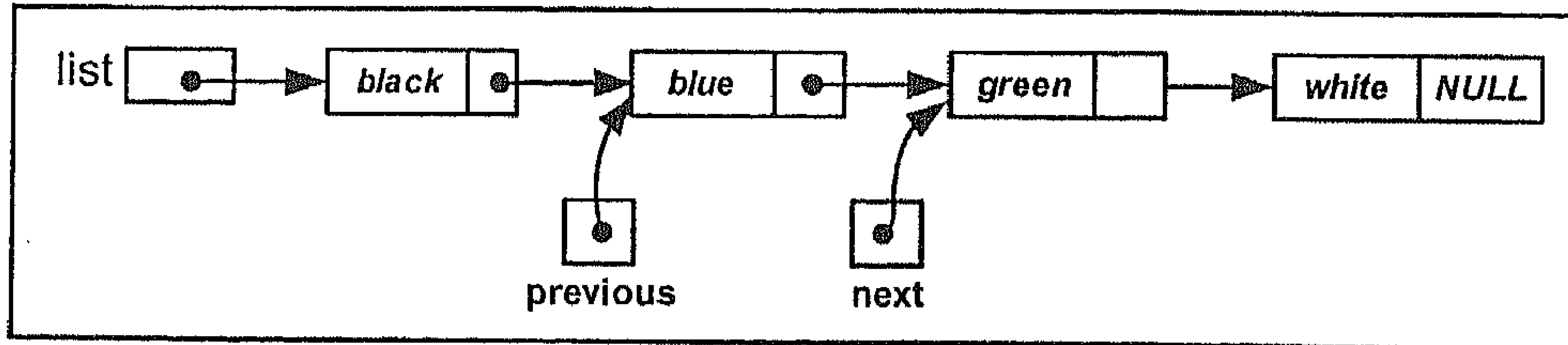
الشكل (9 - د) القائمة بعد إضافة orange

ثم يتلو ذلك استدعاء الدالة locateP لتحديد موقع الإضافة فيتم اختبار الجزء الأول من الشرط، وتكون النتيجة هي false وبالمثل يتم اختبار البديل الآخر من الشرط وتكون النتيجة false لأن القيمة ‘orange’ تأتي بعد القيمة ‘black’ في الترتيب، وبذلك فهي أكبر منها. وعليه، ننتقل إلى else ويتكون لدينا الوضع التالي المبين في الشكل (9 - هـ) لتنفيذ الجملتين اللاحقتين.



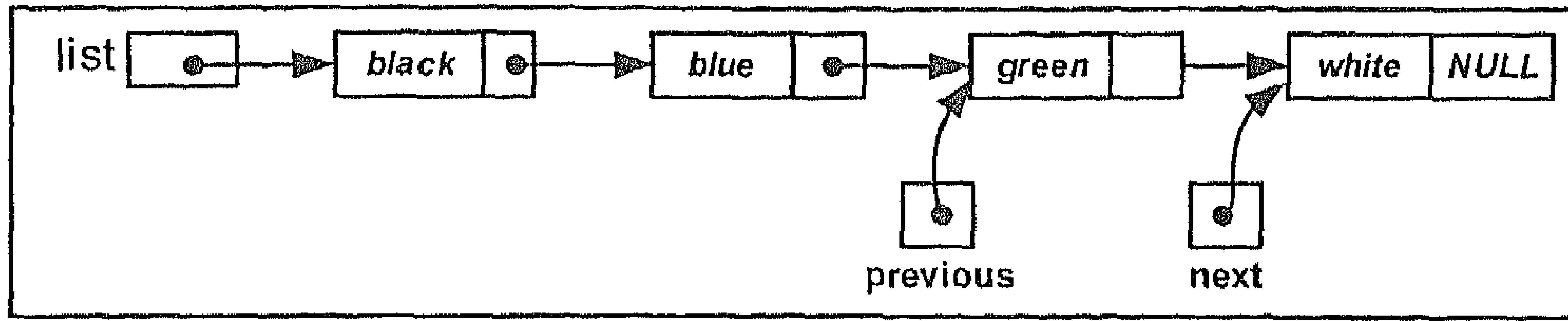
الشكل (9 - هـ): القائمة في حالة البحث عن مكان الإضافة

وبعد ذلك ندخل في التركيب الدوراني حيث يتم اختبار الجزء الأول من الشرط وهو (next == NULL) وتكون النتيجة false ومعنى ذلك هو الانتقال إلى تنفيذ الجملة الأولى والتي تحدد ما إذا كانت ‘orange’ أقل من القيمة المشار إليها بالمؤشر next والإجابة هي بالنفي false . وبذلك ننتقل إلى الجملة التالية لكلمة else ويتم تنفيذ الجملتين اللاحقتين، كما هو مبين في الشكل (9 - و).



الشكل (9 - و): القائمة بعد تحريك المؤشرين إلى الخلف

ثم يتم تنفيذ التركيب الدوراني مرة أخرى ويتم اختبار شرط الدوران وتكون النتيجة مرة أخرى بالنفي ثم يتم اختبار الشرط الوارد في جملة if وتكون النتيجة مماثلة. ثم بالأسلوب نفسه يتم تحريك المؤشرين خطوة أخرى باتجاه الخلف. وبذلك يكون الوضع كما هو موضح في الشكل (9 - ز).



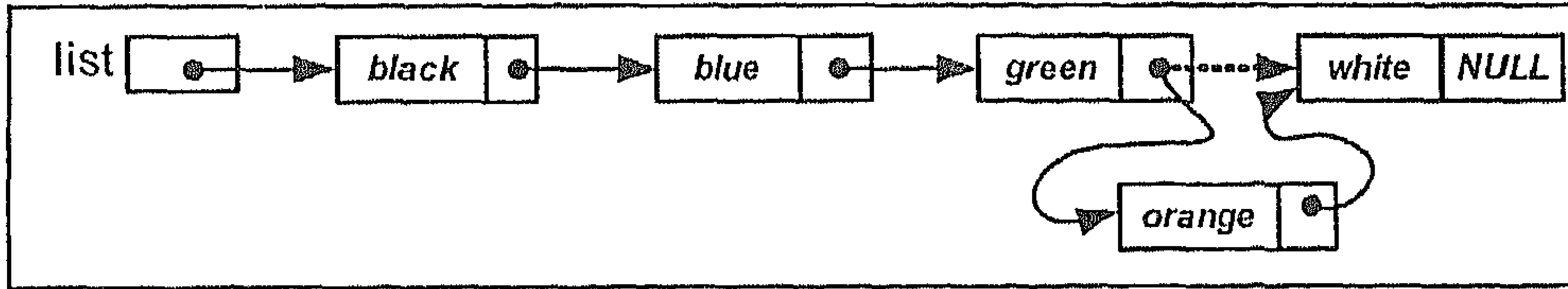
الشكل (9 - ز): القائمة بعد تنفيذ الدوران للمرة الثانية

وللمرة الثالثة يتم تنفيذ التركيب الدوراني وننتقل إلى جملة if فيه وتكون النتيجة بالإيجاب (true) حيث أن القيمة 'orange' أقل بالفعل من القيمة المشار إليها بالمؤشر next وهي 'white' ومن هنا يتم تنفيذ الجملتين

(1) loc = previous; (2) found = true;

وبذلك يتحدد موقع الإضافة، وهو بعد المكان المشار إليه بواسطة loc مباشرة. ومن ثم يتوقف الدوران وينتهي تنفيذ الدالة locateP بكامله، ونعود إلى الدالة insert ومعنا قيمة loc .

والآن يتم تنفيذ الجزء المتبقي من الدالة insert حيث يجري اختبار الشرط المتضمن في if وتكون النتيجة false. ولذلك يتم الانتقال إلى البديل وهو تنفيذ الجمل الواردة بعد else وبذلك نصل إلى الوضع التالي الموضح في الشكل (9 - ح).



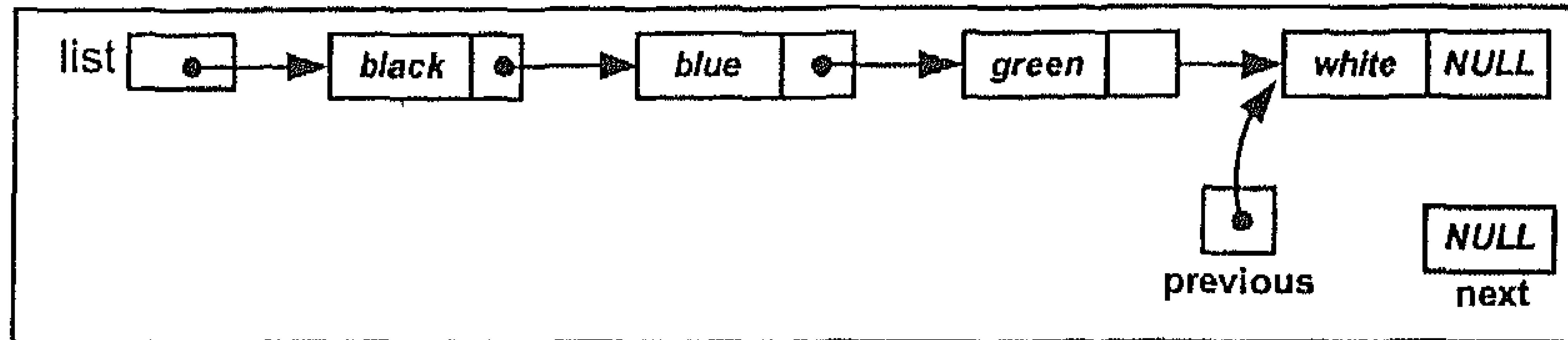
الشكل (9 - ح): وضع القائمة بعد الإضافة

ثالثاً: إضافة العنصر *'yello'*

بقي لدينا العنصر الثالث للإضافة وهو 'yellow' . وكما هو واضح، فإن العنصر يقع في السياق الهجائي بعد 'white' أي في نهاية القائمة. ومعنى ذلك أننا سنمر بجميع الخطوات التي مررنا بها خلال عملية إضافة القيمة 'orange' حتى وصلنا إلى الوضع الممثل في الشكل (9 - ز) . وعليه، يتم تنفيذ التركيب الدوراني للمرة الثالثة، وينتج عن ذلك تحريك المؤشرين باتجاه الخلف مرة أخرى. ولكن، نظراً لأننا وصلنا إلى نهاية القائمة، فإن نتيجة تنفيذ الجملتين الأخيرتين:

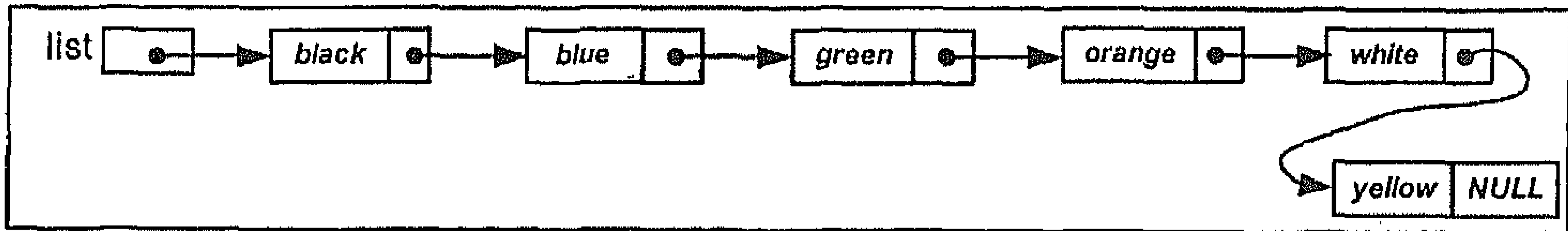
(1) previous = next; (2) next = next-> link;

هي إسناد القيمة NULL إلى المؤشر next وتحويل اتجاه المؤشر previous ليصبح في وضع يشير فيه إلى العنصر الأخير في القائمة على النحو الموضح في الشكل (9 -ط). وطالما أن next قد أصبحت تساوي NULL فإن الدوران ينتهي دون تحديد موقع الإضافة. ثم ننتقل إلى آخر جملة في الدالة locteP وهي: (;loc=previous)، حيث يتم تحديد الموقع بوصفه المكان التالي مباشرة للعنصر الأخير، أي أن الإضافة ستتم في نهاية القائمة.

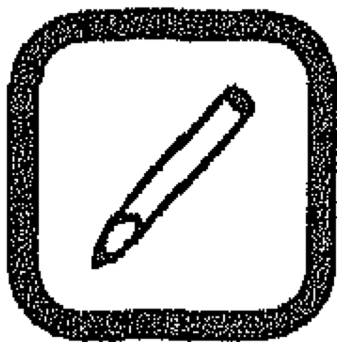


الشكل (9 - ط): البحث عن موقع لإضافة 'yellow'

ثم نعود من الدالة locateP لاستكمال تنفيذ بقية الجمل في insert ويتم ذلك بالأسلوب نفسه الذي تمت به إضافة العنصر المتمثل بالقيمة 'orange'. وبذلك يصبح الوضع النهائي على النحو الموضح في الشكل (9 - ي).



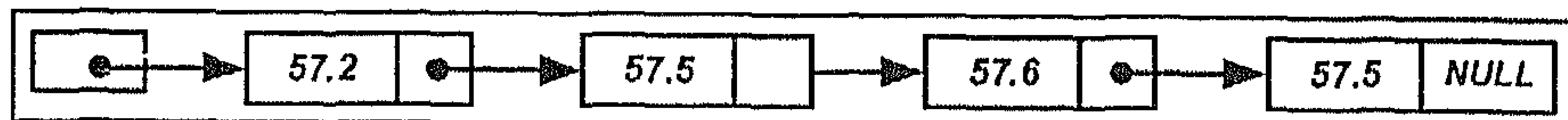
الشكل (9 - ي): القائمة بعد إضافة العنصر الثالث 'yellow'



تدريب (2)

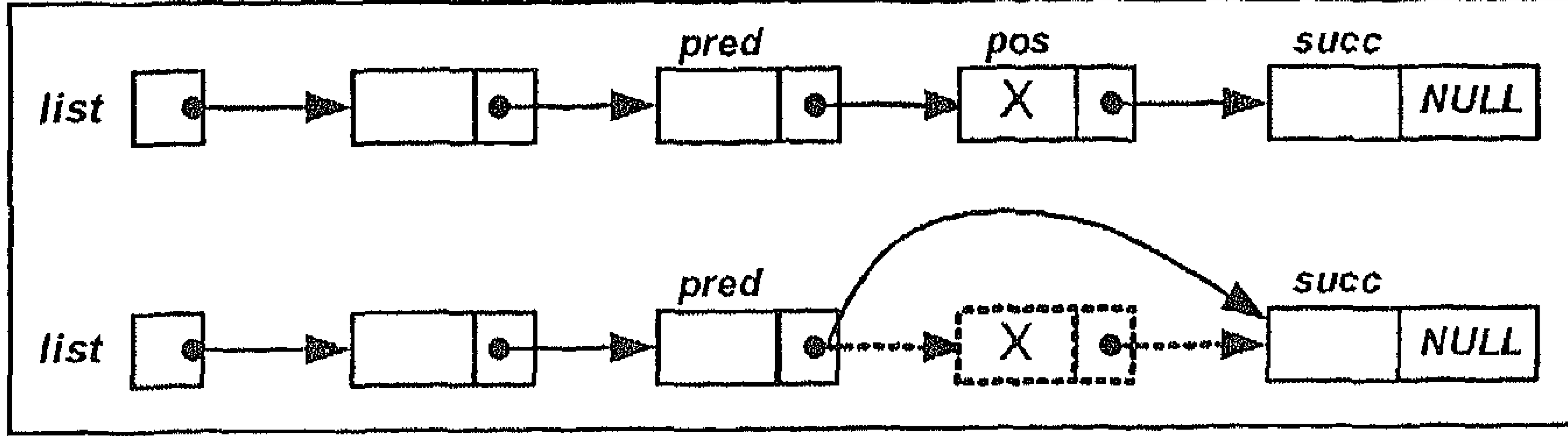
افترض أن لدينا القائمة التالية، ونريد أن نقوم بإضافة القيمة (82.5) في مكانها المناسب ضمن هذه القائمة. فما هو عدد المرات التي ستنفذ فيها كل من الجمل والاختبارات الشرطية التالية:

1. *if (element < next -> info)   // in locateP*
2. *next = next ->link; // in locateP*
3. *newNode -> link = loc-> link   // in insert*



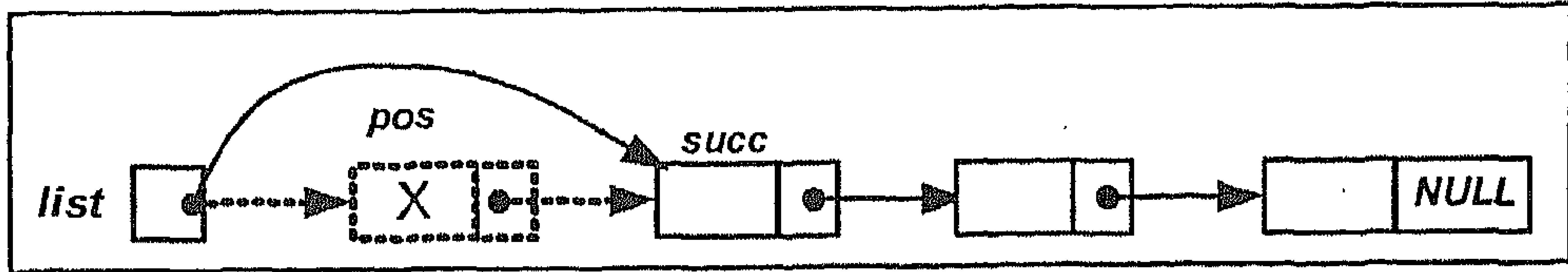
2 - حذف العناصر من القوائم المتصلة (Deletion)

تتم عملية الحذف على العناصر التي لم نعد بحاجة إليها. وتؤدي إلى إجراء بعض التعديل على سلسلة المؤشرات القائمة بين العناصر. شأنها في ذلك شأن عملية الإضافة. فلو أردنا مثلاً أن نقوم بحذف العنصر X من القائمة المتصلة (list) فإن ذلك يقتضي منا أن نحدد أولاً مكان وجود هذا العنصر داخل القائمة ومن ثم نقوم بفك الارتباط بين هذا العنصر وما يليه وما قبله. وإعادة سد الثغرة الحاصلة بواسطة ربط السابق باللاحق، على النحو الموضح في الشكل (10).



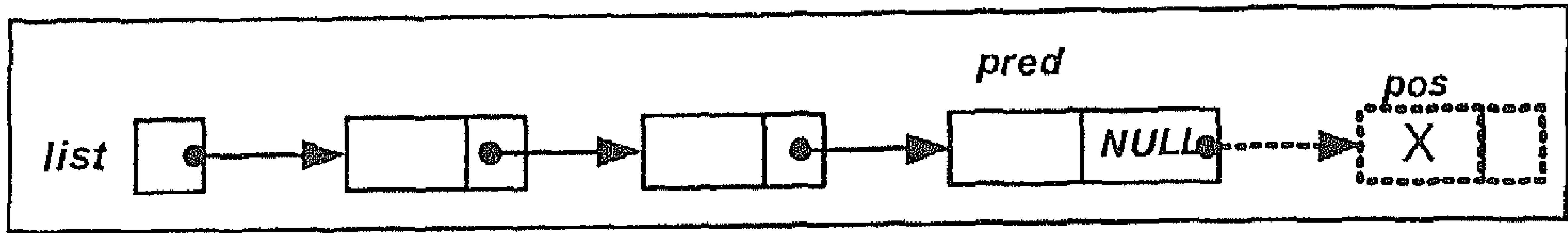
الشكل (10): حذف عنصر واقع بين عنصرين آخرين في القائمة

وكما تلاحظ، فإن العنصر المحذوف في هذا الشكل يقع بين عنصرين، وبذلك فإن عملية الحذف قد انطوت على فك ارتباط العنصر السابق (pred) بالعنصر التالي (succ). والحق أن الحذف يمكن أن يقع في أي مكان في القائمة. فقد يقع العنصر المرغوب حذفه في بداية القائمة، وقد يكون في نهايتها. فإذا كان العنصر المرغوب حذفه في بداية القائمة فإن ذلك يقتضي تعديل المؤشر الخارجي (list) لكي يشير إلى العنصر الثاني في القائمة ثم فك ارتباط العنصر الأول، وذلك على النحو الموضح في الشكل (11). وبذلك فإن العملية لا تنطوي على مجهود كبير.



الشكل (11): حذف عنصر من بداية القائمة

أما في حالة وجود العنصر المرغوب حذفه في نهاية القائمة، فإن ذلك يستدعي فك ارتباط العنصر السابق (pred) بالعنصر الأخير، وذلك بتغيير قيمة المؤشر الخاص بالعنصر المشار إليه بواسطة (pred) إلى NULL . وبذلك تتخذ عملية الحذف الصيغة الموضحة في الشكل (12).



الشكل (12): حذف عنصر من نهاية القائمة

وفيما يلي نبين كيف يمكن أن نقوم بعملية الحذف من خلال الخوارزمية التالية:

الخوارزمية (4):

```
slist* slist::deletee(slist* list ,int element)
  { slist *pred, *posn;
    posn=list;
    pred=findPosn(list,posn,element ) ;
    if (pred==NULL) pred=NULL;
     else
     posn=pred->link;
   if  (posn==NULL)
     cout<< «element is not in LIST|n»;
     else
     if (pred==NULL)// delete first
      list=list->link;
      else
       {
        pred->link = posn->link;
        delete(posn);
        }
   return list;
    }
```

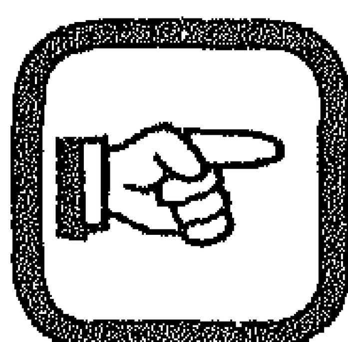
الخوارزمية (5):

```
slist* slist::findPosn(slist* list, slist* posn, int element )
 { slist *current, *back, *pred;
   pred=list;
    if (list==NULL) // list empty
     posn=NULL ;
     else
       if (list->info== element) // first node
        { posn=list;
         pred=0;
         } // end if
```

```
else  // search for location
 {back = list;
  current=list->link;
  while (current !=NULL && current->info !=element)
    {back=current;
     current=current->link;
     } // end while
  pred=back;
  posn=current;

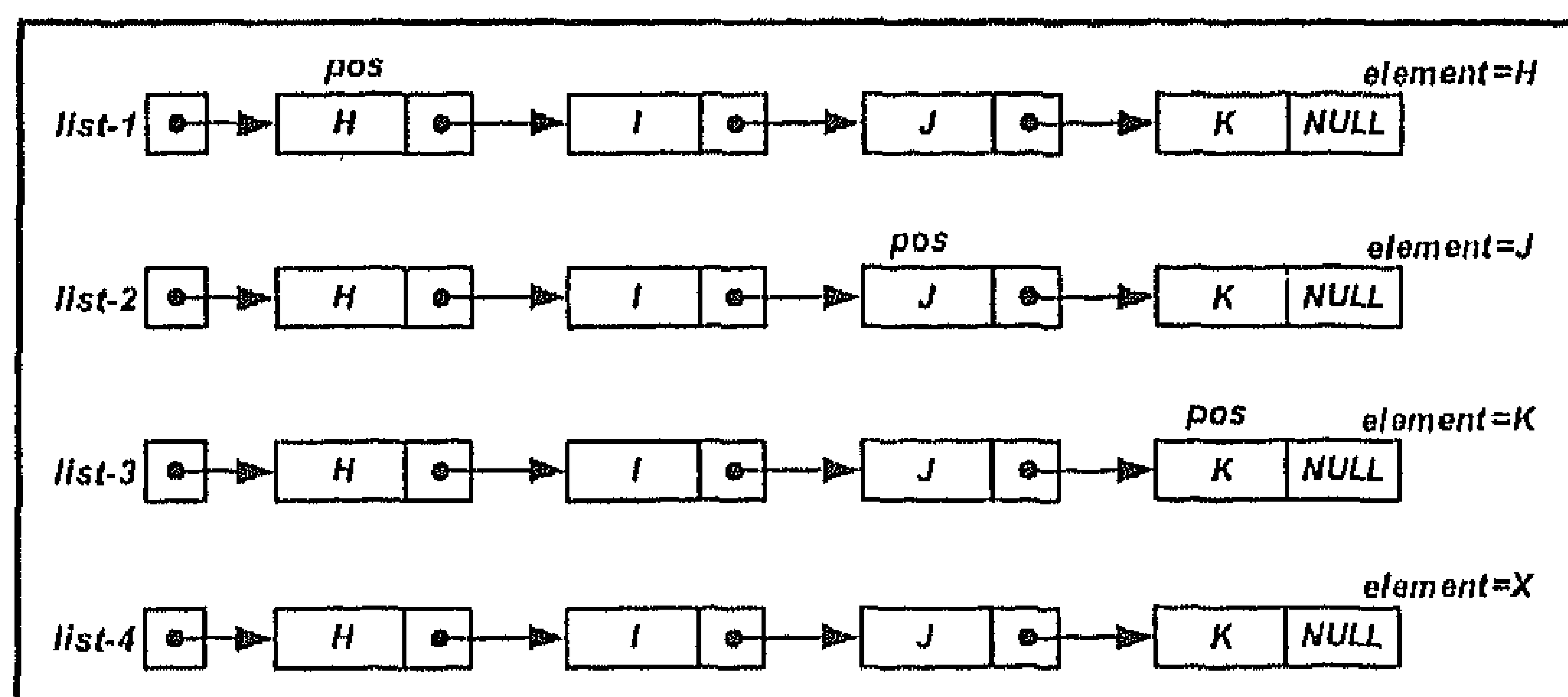
  } // end else
return pred;
} // end  FINDPOSN
```

وكما تلاحظ، فإن خوارزمية الحذف قد استعانت بخوارزمية أخرى للبحث عن موقع العنصر الذي نرغب في حذفه. ونتيجة خوارزمية الاستقصاء (findposn) هي تحديد هذا الموقع والإشارة إليه بالمؤشر الخارجي (posn) والإشارة إلى العنصر السابق عليه بمؤشر خارجي آخر هو (pred) حتى نتمكن من إجراء التعديل المناسب على المؤشرات. فلو اكتفينا فقط بتحديد موقع العنصر المطلوب، لما توفرت لدينا الإمكانية الكافية لسد الفجوة وإعادة ربط قافلة العناصر ببعضها. والآن دعنا ننظر إلى المثال التالي لنرى كيف يمكن تطبيق خوارزمية الحذف.



مثال (4)

افترض أن لدينا القائمة المتصلة التالية الموضحة في الشكل (13) والتي تظهر في أربعة أوضاع مختلفة. وافترض أننا نرغب في حذف العنصر المشار إليه بالمؤشر الخارجي (posn):



الشكل (13): الأوضاع المختلفة للحذف

كما تلاحظ، فإن هذا العنصر يقع في بداية القائمة في الوضع الأول للقائمة (list-1) ويأتي في داخل القائمة في الوضع الثاني (list-2). ويأتي في نهاية القائمة في الوضع الثالث (list-3). ولا يظهر في القائمة مطلقاً في الوضع الرابع (list-4).

أولاً: حذف عنصر من بداية القائمة

دعنا نبدأ أولاً بتطبيق الخوارزمية على الوضع الأول في الحذف. عند تنفيذ خوارزمية الحذف والبدء بجملة استدعاء (findposn) لإيجاد موقع العنصر الذي نرغب في حذفه، عندها ينتقل التحكم إلى الخوارزمية الثانية حيث يتم ما يلي:

أ. *if (list==NULL)* والإجابة بالنفي، ومعنى ذلك الانتقال إلى البديل else.

ب. *if (list->info== element)* والإجابة بالإثبات، ومعنى ذلك أن العنصر واقع في بداية القائمة.

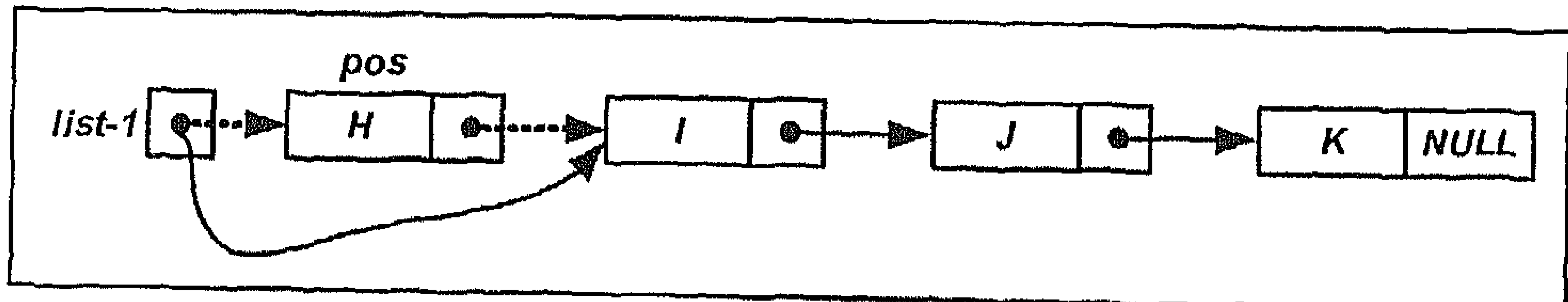
ج. *posn= list* أي أن المؤشر *posn* يشير الآن إلى العنصر المطلوب.

د. *pred = NULL* طالما أن العنصر المطلوب هو الأول في القائمة فإنه ليس له سابق وبالتالي فإن قيمة *pred* هي *NULL*.

وبذلك ينتهي تنفيذ الخوارزمية (findposn) ونعود إلى الخوارزمية (deletee) حيث يستأنف التنفيذ ابتداء من الجملة التالية للاستدعاء، حيث يتم ما يلي:

1. *if (posn==NULL)* والإجابة هي بالنفي، ولذلك يتم تنفيذ البديل else.
2. *if (pred==NULL)* والإجابة بالإثبات، وبذلك يتم تنفيذ الجملة التالية.
3. *list=list->link;* يتم فك ارتباط المؤشر الخارجي list بالعنصر المراد حذفه ويحول اتجاهه إلى العنصر التالي له.
4. *delete(posn);* حذف العنصر تماماً وإلغاء الحجز لهذا العنصر من الذاكرة.

ويبين الشكل (14) حذف عنصر من بداية القائمة list-1 الواردة في الشكل (13) نتيجة للقيام بعملية الحذف.



الشكل (14): حذف عنصر من بداية القائمة (posn= list)

ثانياً: حذف عنصر واقع بين عنصرين آخرين في القائمة

افترض مرة أخرى أن القائمة المتصلة في وضعها الثاني (list-2) ونريد أن نحذف العنصر المتمثل بالقيمة 'j'. وهذا العنصر، كما ترى، واقع بين عنصرين آخرين داخل القائمة. إذن عند البدء بتنفيذ جملة الاستدعاء وينتقل التحكم بسير العمليات إلى الخوارزمية (5)، حيث يتم ما يلي:

أ. *if (list==NULL)* والإجابة بالنفي، ومعني ذلك الانتقال إلى البديل else.

ب. *if (list->info== element)* والإجابة بالنفي، معنى ذلك الانتقال إلى البديل else. حيث يتم البحث عن مكان وجود العنصر في داخل القائمة.

ج. *back = list;* المؤشر المحلي (back) يشير إلى بداية القائمة الآن.

د. *current = list-> link;* المؤشر المحلي (current) يشير إلى العنصر الثاني في القائمة، (انظر الجزء الأول من الشكل 15).

هـ. *while ... do* يتم اختبار الشرط المركب للدخول في التركيب الدوراني، والنتيجة هي بالإيجاب. وبذلك يتم تنفيذ الجمل الداخلية التالية.

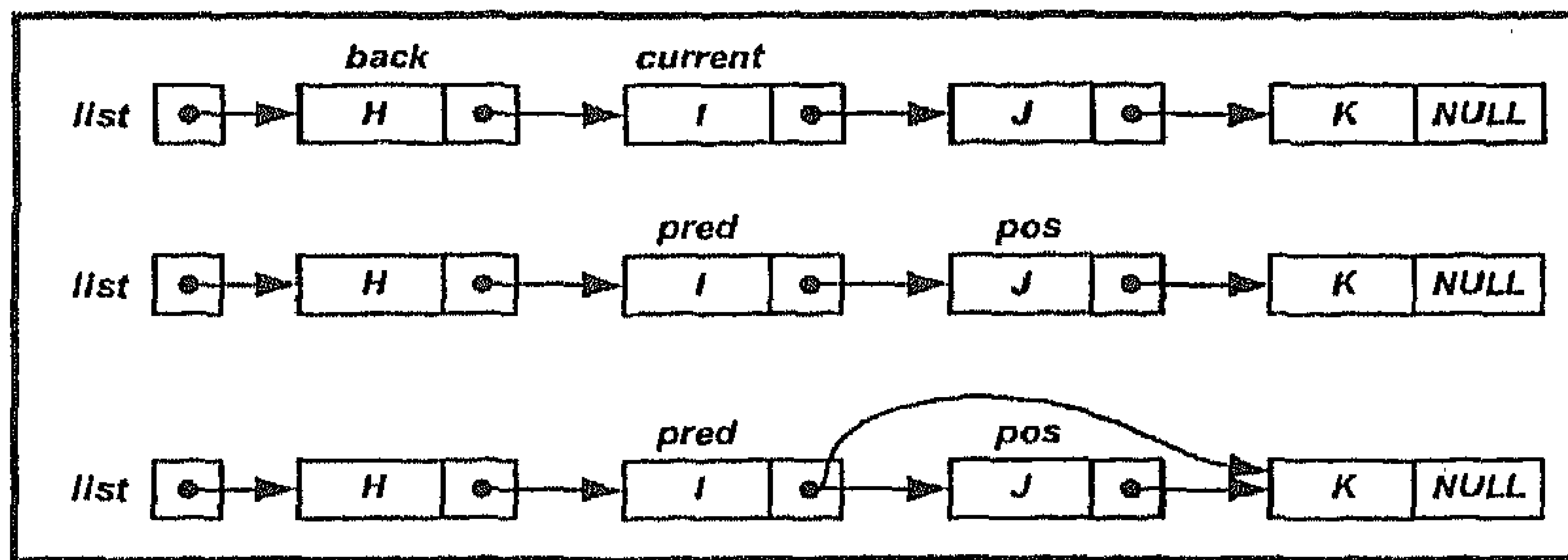
و. *back= current;* يتحرك المؤشر (back) خطوة باتجاه الخلف.

ز. *current= current-> link;* يتحرك المؤشر (current) خطوة باتجاه الخلف. (انظر الجزء الثاني من الشكل 15).

ح. *while ... do* يتم اختبار شرطي الدوران مرة أخرى، والنتيجة هذه المرة بالنفي نظراً لأن (current->info) مساوية الآن لقيمة العنصر الذي نبحث عنه وهو 'j' وبذلك نخرج من الدوران وننتقل إلى الجمل التالية للتركيب الدوراني.

ط. *pred= back* المؤشر pred يشير الآن إلى العنصر السابق.

ي. *posn= current* المؤشر posn يشير إلى موقع الحذف (انظر الجزء الثاني من الشكل 15)



الشكل (15): القائمة بعد حذف العنصر المتمثل بالقيمة 'j' من القائمة

والآن بعد أن حددنا مكان العنصر المرغوب، نعود إلى الدالة (delete) حيث يتم ما يلي:

أ. *if (posn= =NULL)* والإجابة هي بالنفي، وبذلك ننتقل إلى البديل else .

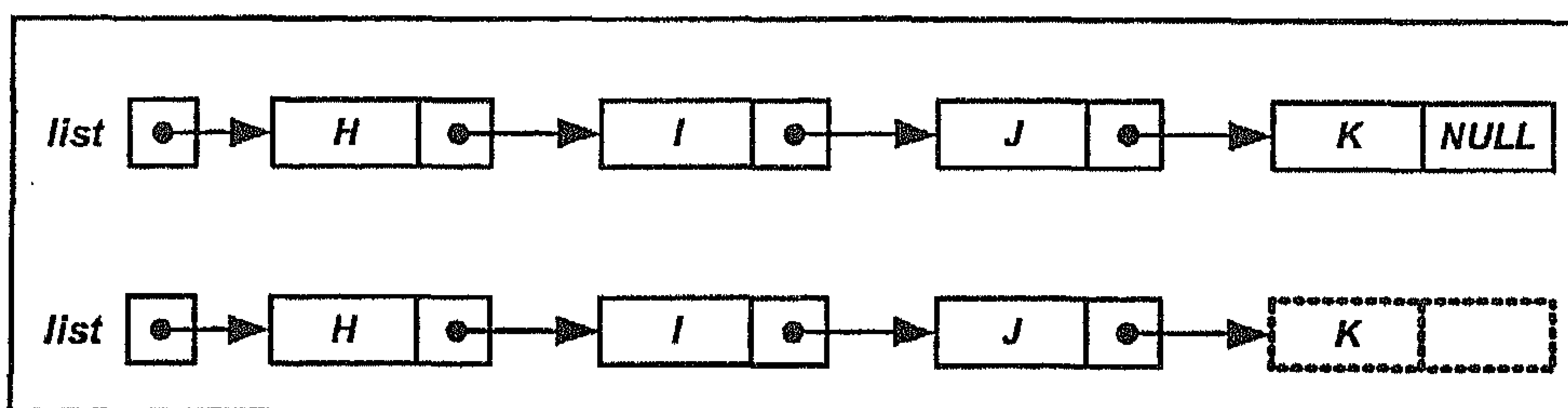
ب. if (pred= =NULL) والإجابة مرة أخرى بالنفي، وبذلك ننتقل إلى else .

ج. pred->link=pos->link; يتم فك ارتباط العنصر السابق بالعنصر المرغوب حذفه ويحول اتجاه المؤشر إلى العنصر التالي (انظر الجزء الثالث من الشكل (15)).

د. delete (pos); يتم التخلص من العنصر بإلغاء الحجز من الذاكرة.

ثالثاً: حذف عنصر من نهاية القائمة

افترض مرة ثالثة أن القائمة المتصلة في وضعها الثالث الآن (list-3) ونريد أن نحذف العنصر المتمثل بالقيمة "k" . فكما تلاحظ من الشكل (13) هذا العنصر يقع في نهاية القائمة، ومعنى ذلك أننا سنمر بجميع الخطوات التي مررنا بها خلال تنفيذ الخوارزمية (findpos) من أجل حذف العنصر المتمثل بالقيمة 'j' ولكن عندما نصل إلى الخطوة "ح" أعلاه فإن شرط الدوران يتحقق للمرة الثانية وبذلك يتم تحريك المؤشرين المحليين خطوة أخرى إلى الخلف. وبذلك يصبح المؤشر (current) في وضع يشير فيه إلى آخر عنصر في القائمة وهو العنصر المقصود. وبذلك لن يتحقق شرط الدوران للمرة الثالثة، ويتم تنفيذ الجمل التالية له، ويتحدد موقع العنصر المطلوب على النحو الموضح في الخطوتين "ط" و"ي" أعلاه. والجزء الأول من الشكل (16) يوضح الوضع الذي انتهت إليه القائمة بعد الانتهاء من تنفيذ (findpos).

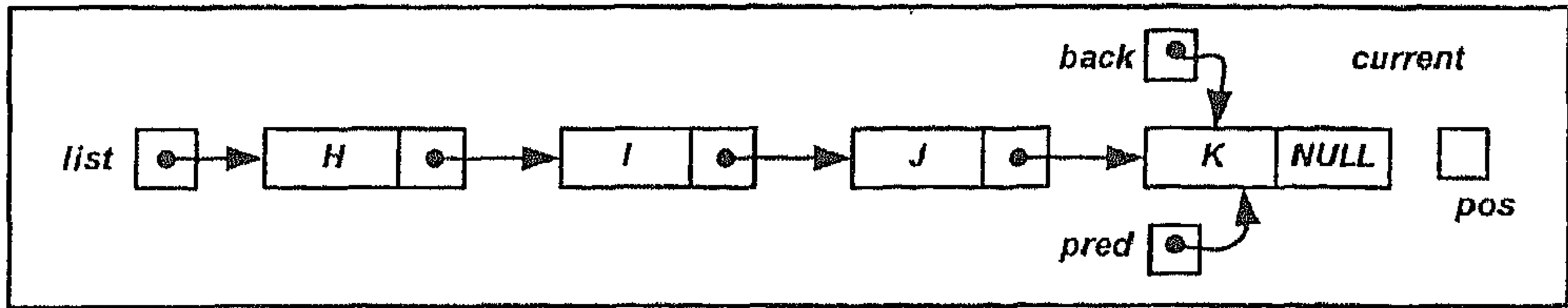


الشكل (16): القائمة بعد حذف العنصر الأخير منها

وبعد أن نعود إلى الدالة (delete) فإن تنفيذ الاختبارات الموجودة سيؤدي بنا إلى السياق نفسه من الخطوات التي مررنا بها خلال حذف العنصر المتمثل بالقيمة "j" أعلاه. وبالتالي تصبح قيمة العنصر الأخير في القائمة NULL على النحو الموضح في الجزء الثاني من الشكل (16).

رابعاً: حذف عنصر غير موجود في القائمة

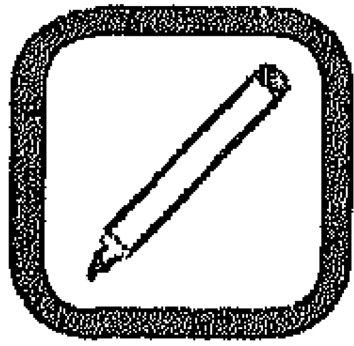
أخيراً، نحن أمام وضع نبحث فيه عن العنصر المراد حذفه (وهو x في هذه الحالة) ولا نجده في القائمة. ومعنى ذلك أننا سنمر بجميع الخطوات التي مررنا بها خلال بحثنا عن موقع العنصر "k" أعلاه. وبالإضافة إلى ذلك، سيتم تنفيذ التركيب الدوراني المتضمن في الدالة (findpos) مرة إضافية. وبذلك يتحرك المؤشران المحليان خطوة أخرى باتجاه الخلف، مما يؤدي إلى أن يصبح الوضع على النحو الموضح في الشكل (17). فقيمة المؤشر (current) هي null وقيمة المؤشر (back) هي الإشارة إلى آخر عنصر في القائمة.



الشكل (17): وضع القائمة بعد انتهاء عملية البحث عن العنصر x

وعند العودة إلى الدالة (delete) فإن جواب الشرط *if (posn==NULL)* سيكون الإثبات، وبالتالي فإننا سنتلقى الإشعار التالي القائل أن العنصر غير موجود في القائمة:

Element is not in list



تدريب (3)

تأمل القائمة المعطاة في التدريب (2) المذكور سابقاً، وافترض أننا نرغب في حذف العنصر الأخير المتمثل بالقيمة (90.0) فما هو عدد مرات تنفيذ الجمل والاختبارات الشرطية التالية:

1. *current= current-> link;* *//in finspos*
2. *if (list-> info = element)* *//in findpos*
3. *if (pred== NULL)* *//in delete*

3.3 تمثيل القوائم بالمصفوفات الأحادية
Array implementation of lists

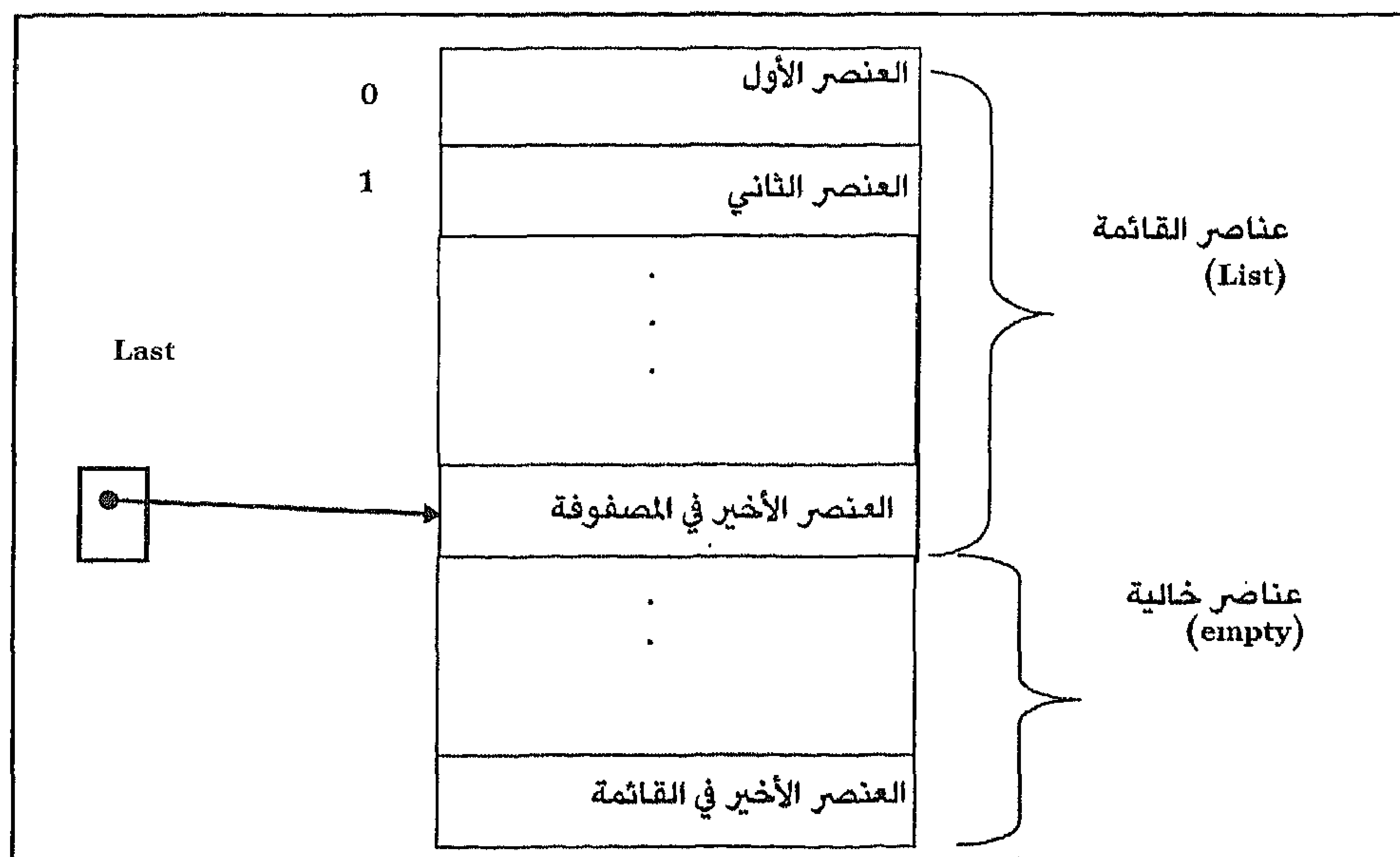
سبق أن صادفت المصفوفات ذات البعد الواحد في أكثر من موضع وفي أكثر من مناسبة، خارج نطاق هذا المقرر. ولا بد أنك تذكر بأنك قد درست موضوع المصفوفات من وجهة نظر برمجية، بالتفصيل في لغة سي/سي++. ولا نريد أن نكرر ما درسته هناك، فالحديث هنا يتخذ طابعاً مختلفاً عما درسته في أي مقرر آخر.

فالمصفوفة الأحادية، كما تعلم مجموعة متجانسة من العناصر تتخذ شكلاً خطياً، وتخزن في مواضع متجاورة في ذاكرة الحاسوب. وبناء على ذلك فإن العنصر (i) في المصفوفة يأتي مسبوقاً بالعنصر (i-1) (تقرأ من اليسار إلى اليمين) ويأتي متبوعاً بالعنصر (i+1).

ولعلك تعرف، عزيزي الدارس، بأن الغالبية العظمى من اللغات الراقية توفر نمطاً بيانياً متميزاً للمصفوفات، بما في ذلك المصفوفات الأحادية. ولكن ينبغي أن نشير في هذا المقام إلى أن هذه اللغات (باستثناء لغة الجول - 60) تسير على نهج الحجز الثابت (fixed allocation) للمصفوفة. وبذلك فإن الوجود الكامل للمصفوفة سابق على وجود القيم نفسها. فبينما يتم تثبيت عدد عناصر المصفوفة خلال مرحلة الترجمة، فإن القيم نفسها يتم إسنادها خلال مرحلة التنفيذ، وتزيد هذه القيم وتنقص وتتغير تبعاً للحاجة. وعلى هذا الأساس ينبغي أن نميز بين المصفوفة من الناحية الفيزيائية المتصلة بالذاكرة، ومجموعة القيم التي تحتويها والتي تشكل ما نشير إليه بمصطلح القائمة الملتزة. فقد يكون عدد عناصر القائمة مساوياً لعدد الأماكن المحجوزة، وقد يكون أقل من ذلك.

عندما تحدثنا عن القوائم المتصلة المفردة في القسم السابق أشرنا إلى وجود طريقتين لتمثيل هذا النوع من القوائم: الأولى هي استخدام المؤشرات التي توفرها لغة البرمجة، والثانية هي استخدام الدالات النسبية في المصفوفات المتوازية أو المصفوفات المركبة. والآن نحن بصدد طريقة رئيسة ثالثة مختلفة لا تستخدم المؤشرات من أي نوع. وبدلاً عن ذلك فإنها تخزن مجموعة القيم التي تضمها القائمة في مواضع متجاورة (contiguous, adjacent) ضمن منطقة تم حجزها خصيصاً لهذه الغاية، وكل عنصر يعرف بعنوانه النسبي (أي نسبة إلى بداية المصفوفة) الذي تتم ترجمته من جانب نظام لغة البرمجة المستعملة إلى عنوان مطلق (absolute address) حين البحث عنه في الذاكرة.

وتقوم هذه الطريقة على وجود مصفوفة أحادية بعدد محدد من العناصر، ووجود مؤشر إلى العنصر الأخير في القائمة كما هو موضح في الشكل (18) وهو مؤشر على الحجم الفعلي للقائمة (أي عدد عناصرها) والذي قد يكون أقل من حجم المصفوفة ذاتها. ويمكن أن يكون هذا المؤشر جزءاً من تعريف القائمة، كما يمكن أن يكون تعريفه مستقلاً. وفيما يلي نبين الفرق بين هذين النهجين:



الشكل (18): قائمة مخزنة داخل مصفوفة أحادية

أولاً: المؤشر كجزء من تعريف القائمة

```
const MaxLength=100;  //some suitable constant
  class  ListType {
    int elements [MaxLength];
    int last; //pointer to last element
 };
```

ثانياً: المؤشر كمتغير مستقل

```
const MaxLength=100;  //some suitable constant
int  List [MaxLength];
int last; //pointer to last element
```

وكما تلاحظ، فإن الأسلوب الأول يقوم على فكرة الصنف (class). وهذا الصنف يتضمن حقلين: أحدهما هو لمصفوفة أحادية والآخر هو المؤشر (last). أما في الأسلوب الثاني، فإن تعريف المتغير (last) قد جاء مستقلاً عن تعريف المصفوفة. وينبغي التنبه إلى أن الاختلاف بين الأسلوبين يؤدي إلى اختلافات في صيغة الإشارة إلى عناصر القائمة داخل البرنامج، ولكنه لا يؤدي إلى اختلاف في الوظيفة والنتيجة.

ونحن نفضل الأسلوب الأول وذلك لأننا لو استخدمنا الأسلوب الثاني فسيعني ذلك تغير ترويسة العمليات التي عرفناها على القوائم مسبقاً. ستتغير ترويسة العمليات لإضافة العامل (Parameter) (last). ولكن استخدام الأسلوب الأول لا يتطلب ذلك الأمر وعليه فإننا سنعتمد الأسلوب الأول فيما تبقى من هذه الوحدة.

وأياً كانت صيغة التعريف، فإن من المسائل الهامة بالنسبة للمصفوفات تحديد عنوان الموضع الذي تم تخزين أحد العناصر فيه. فإشارة المبرمج إلى عنصر معين من عناصر المصفوفة بالطريقة المعروفة وهي استخدام اسم المتغير متبوعاً بالدالة النسبية. (مثال ذلك: [5]a)، وهي إشارة منطقية تدل على عنوان نسبي ولا بد من قيام الجهاز بالنهاية بترجمتها إلى عنوان مطلق، كما أشرنا قبل قليل.

ويعتمد القيام بهذه المهمة على توفر معلومتين: المعلومة الأولى هي العنوان المطلق لبداية المصفوفة (أي عنوان الموضع الأول في المساحة التي تم حجزها للمصفوفة في الذاكرة) ونطلق على هذا العنوان اسم الأساس (base) والمعلومة الثانية هي مقدار المسافة بين الأساس والعنصر المطلوب، فيما نشير إليه باسم البعد الطولي (offset). وتتم عملية الحساب هذه على النحو التالي:

فلو افترضنا أن لدينا مصفوفة معرفة بالصيغة التالية:

int a[10];

وافترضنا أن الأساس هو (2000) وأن كل قيمة من القيم المخزنة في هذه المصفوفة تحتاج إلى موضع واحد (word) فإن شكل تخزين المصفوفة داخل الذاكرة سيكون على النحو الموضح في الشكل (19). وعلى هذا الأساس فإن العنوان المطلق للعنصر الخامس في المصفوفة [5]a داخل الذاكرة هو: (2005). وقد وصلنا إلى هذه النتيجة من خلال النظر إلى الشكل (19). ولكن لا بد من أسلوب منهجي للوصول إلى ذلك، وهو يقوم على المعادلة التالية:

O (a[I]) = base + size (offset)

حيث أن:

base = العنوان المطلق للموضع الأول في المصفوفة (الأساس)

size = حجم المكان المخصص لكل عنصر

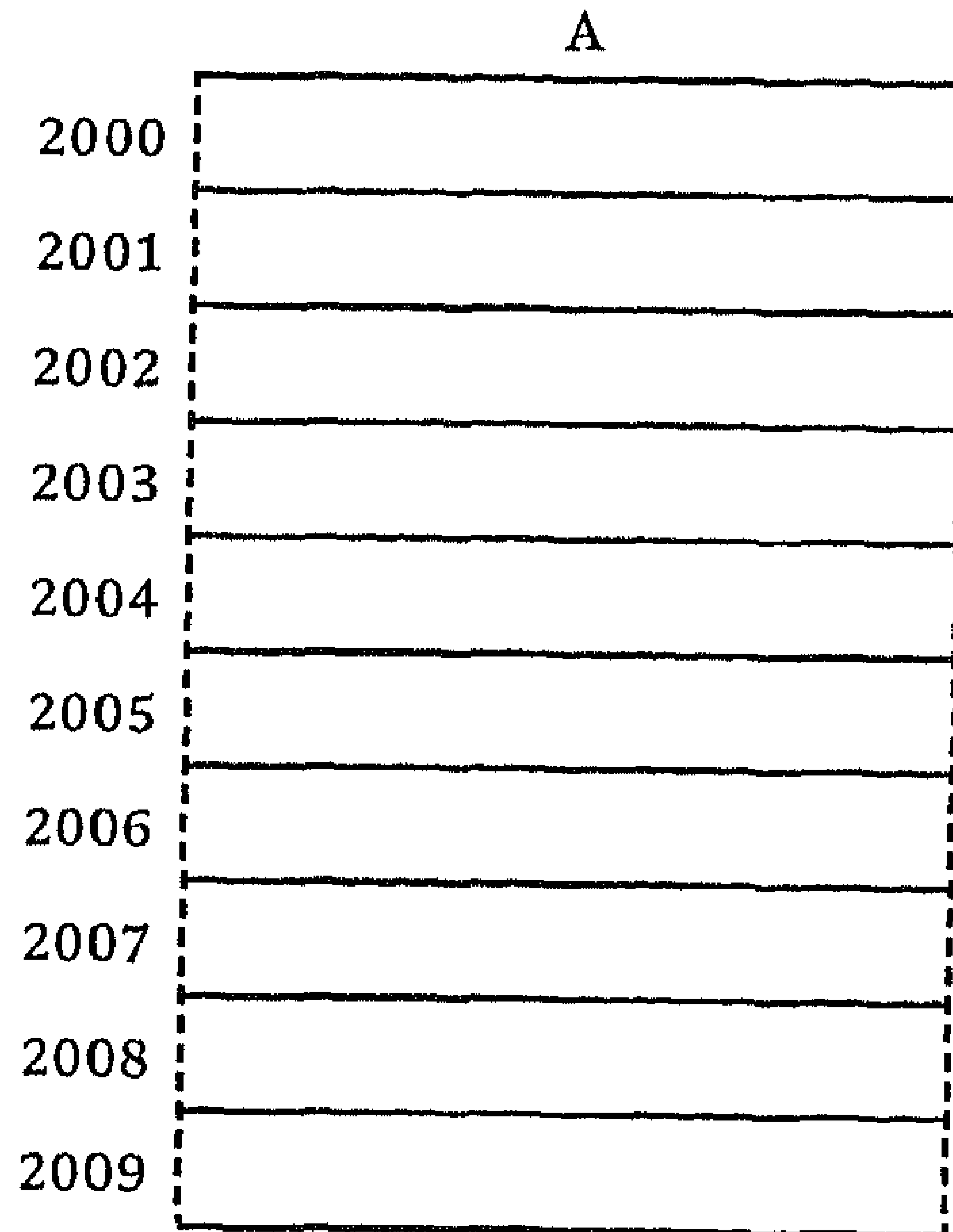
(I-lowerbound) = offset

وعليه فإن العنوان المطلق (الفعلي)

للعنصر[5]a هو:

O (a [5] = 2000 + 1 (5 - 0)

2005 =



الشكل (19): طريقة تمثيل المصفوفة في الذاكرة

إن هذه الطريقة في تمثيل القوائم تتيح للمبرمج إمكانية استعراض العناصر بقدر كبير من السهولة، كما أن إضافة عناصر جديدة إلى نهاية القائمة تتم بنفس القدر من البساطة والمرونة. ولكن في مقابل ذلك ينبغي أن نذكر بأن إضافة قيمة جديدة إلى منتصف القائمة تتطلب إزاحة جميع القيم التالية لها إلى المكان المجاور باتجاه مؤخرة المصفوفة حتى يتيسر إيجاد مكان للقيمة المضافة. وبالمثل فإن حذف القيم من القائمة فيما عدا الأخيرة، يؤدي إلى إزاحة جميع العناصر التالية لها إلى الموضع المجاور باتجاه مقدمة المصفوفة لسد الثغرة التي نتجت عن الحذف، كما سنرى عند الحديث عن العمليات التي تجري على المصفوفات الأحادية.

ومن هنا، يبرز لدينا سؤال يستحق الاهتمام وهو: أي الطرق المستعملة في تمثيل القوائم أفضل؟ هل نختار طريقة استخدام المؤشرات التي توفرها اللغة (مثل لغة سي/ سي++) أم طريقة الدالات النسبية الخاصة بالمصفوفات المتوازية أو المصفوفات المركبة، أم طريقة المصفوفات الأحادية؟ وللإجابة على هذا السؤال نقول أن هناك مجموعة من المسائل ينبغي أخذها بالحسبان، وهي:

أ. إن التمثيل بالمصفوفات، على اختلاف تطبيقاتها، يتطلب منّا أن نقرر سلفاً، منذ وقت الترجمة الحد الأعلى لعدد العناصر. فإذا لم يكن باستطاعتنا تحديد هذا العدد، فإن من الأفضل أن نختار طريقة التمثيل بالمؤشرات إذا كانت اللغة توفر مثل هذه الإمكانية.

ب. تتفاوت الأساليب الثلاثة فيما بينها من حيث الوقت الذي تستغرقه بعض العمليات. فبينما تتطلب عمليتا الإضافة والحذف، على سبيل المثال، في القائمة المتصلة (على اختلاف أساليب تمثيلها) عدداً ثابتاً من الخطوات، فإنها تستغرق وقتاً يتناسب مع عدد القيم التي ينبغي إزاحتها في حالة المصفوفات الأحادية. وفي المقابل فإن عملية تحديد العنصر السابق والعنصر الأخير في المصفوفات الأحادية تستغرق وقتاً ثابتاً، بينما تستغرق العملية وقتاً يتناسب مع عدد العناصر في حالة القائمة المتصلة.

ج. إن القوائم الخطية المفردة لا تتمتع بقدر مناسب من المرونة. فلو تم تحديد موضع معين للإضافة أو الحذف بواسطة مؤشر خارجي فإن هذا المؤشر لن يصلح لغير الحالة التي أسند لها. لأن هذا المؤشر، بحكم طبيعة تكوينه، يستطيع أن يتحرك باتجاه واحد فقط. والأسلوب الوحيد لتجاوز هذه المشكلة هو استخدام صيغة القائمة الثنائية، وبذلك يمكن توفير قدر مناسب من المرونة وإن كان لا يصل لدرجة المرونة نفسها التي تتمتع بها عملية الإشارة إلى العناصر في حالة المصفوفات الخطية.

د. إن أسلوب التمثيل بالمصفوفات على اختلاف أنواعها، أحادية أو متوازية أو مركبة، تؤدي إلى إهدار في المساحة. وذلك لأن عدد عناصر المصفوفة مستقل عن عدد القيم. أما في حالة التمثيل باستخدام المؤشرات، فإن الحجز في الذاكرة يتم بناء على الطلب. لكن الثمن الذي ندفعه مقابل ذلك هو جزء إضافي من الذاكرة للاحتفاظ بالمؤشرات. وبذلك يمكن القول، بصفة عامة، بأن أسلوب المؤشرات أكثر استثماراً للذاكرة. ولكن ينبغي التنبيه إلى أن هناك بعض الظروف التي يمكن أن تكون فيها المصفوفات أكثر استثماراً للذاكرة، خاصة في حالة الأعداد الثابتة من القيم.

أ. تنفيذ عمليات (OPERATIONS) القوائم باستخدام المصفوفات

إن جميع العمليات التي أشرنا إليها في الوحدة الأولى من هذا المقرر وفي بداية هذه الوحدة يمكن أن تتم على القوائم التي تمثل على شكل مصفوفات أحادية. إذ من الممكن أن نقوم بعمليات الاستعراض، والاستقصاء، والإضافة، والحذف، والفرز، وبعض العمليات الأخرى. ولكن نظراً لأن هناك وحدة مستقلة من هذا المقرر مخصصة لمناقشة المسائل المتعلقة بالاستقصاء والفرز، فإننا لن نتطرق لهاتين العمليتين هنا. ونكتفي بشكل خاص بعمليات الاستعراض، والإضافة، والحذف.

إنشاء القائمة Listcreate

في حالة استخدام المصفوفات لتمثيل القائمة فإن كل ما تعمله هذه الدالة هو إعطاء المؤشر last القيمة صفر

```
void  MtList::Listcreate(MtList& L)
  {  L.last= 0;
   }
```

الاستعراض *(array-list traversal)*

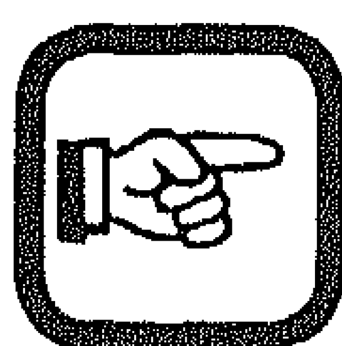
كما تعلم، عزيزي الدارس، فإن مجرد تعريف المصفوفة ليس كافياً بحد ذاته. فلا بد من إسناد مجموعة من القيم إلى هذه المصفوفة باستخدام أحد التراكيب الدورانية المعروفة في لغات البرمجة مثل: (while) أو (do-while) أو (for). ولا يعني ذلك، بالضرورة أن نملأ جميع خلايا المصفوفة بالقيم. فعدد القيم الفعلي يمكن أن يزيد أو يقل عن العدد الكلي للخلايا المحجوزة. وبذلك فإن العمليات التي تتم على المصفوفات ينبغي أن تكون محدودة بعدد القيم المخزنة في المصفوفة، والتي نشير إليها من خلال متغير خاص، تحدثنا عنه في الجزء السابق.

ومن هنا فإننا عندما نقوم بعملية الاستعراض لا نتعامل مع الحد الأعلى للمصفوفة، بل مع المتغير الذي يشير إلى آخر قيمة مخزنة في المصفوفة (وهو last في الخوارزمية 6). فعلى سبيل المثال، لو كان عدد عناصر المصفوفة عشرين عنصراً، وعدد القيم المخزنة فيها هو سبعة، فإن عملية الاستعراض ستقتصر على العناصر التي تشكل قائمة القيم فقط. وفيما يلي نورد الخطوات التي تنطوي عليها هذه العملية.

الخوارزمية (6):

```
void  MtList::Traversal (MtList list)
  {//traversing an array-based list
   for (int i= 0; i<list.last;i++)
  someAction(list.elements [i]);
   } // traversal
```

وكما تلاحظ، فإن هذه الخوارزمية تميل إلى أسلوب التجريد من حيث طبيعة المعالجة التي تجري على البيانات. وقد سبق أن أشرنا من قبل إلى أن هذه العملية (someAction) يمكن أن تعبر عن طباعة القيم أو تعدادها أو البحث عن أصغر أو أكبر قيمة في مجموعة غير منظمة من القيم. وفيما يلي نورد مثالاً على ذلك.



مثال (5)

افترض أن لدينا المصفوفة التالية الموضحة في الشكل (20) ونرغب في:

أولاً: طباعة جميع القيم الموجودة في المصفوفة.

ثانياً: حساب عدد القيم التي تقل عن 50.

فإن ذلك يمكن أن يتم، في ضوء الخوارزمية (6)، على النحو التالي:

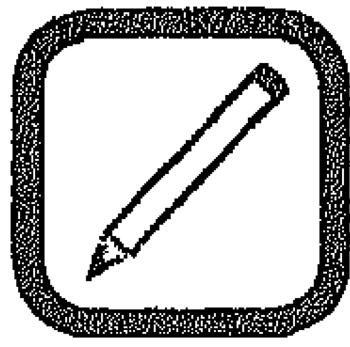
0	1	2	3	4	5	6	7	8	9	10	11	12	13
70	45	67	84	42	87	35	91	75					

الشكل (20): مصفوفة خاصة بالمثال (5)

```
1.  void  MtList::PrintValues(MtList    list )
     { for (int i=0; i<list.last;i++)
     cout<<list.elements [i]<<" ";
     }

2.   void  MtList::countValues(MtList    list )
      {int count=0;
        for (int i=0; i<list.last;i++)
          if (list.elements[i]< maxlength ) count++;
        cout<<"count="<< count<<endl;
      }
```

تدريب (4)



تأمل المصفوفة الموضحة في الشكل (20) مرة أخرى ثم أعد كتابة خوارزمية الاستعراض مرة ثانية بالأسلوب الذي وضحناه في المثال (5) وذلك من أجل استخراج متوسط القيم المذكورة.

ب - إضافة قيم جديدة إلى المصفوفة الأحادية (inserting new values)

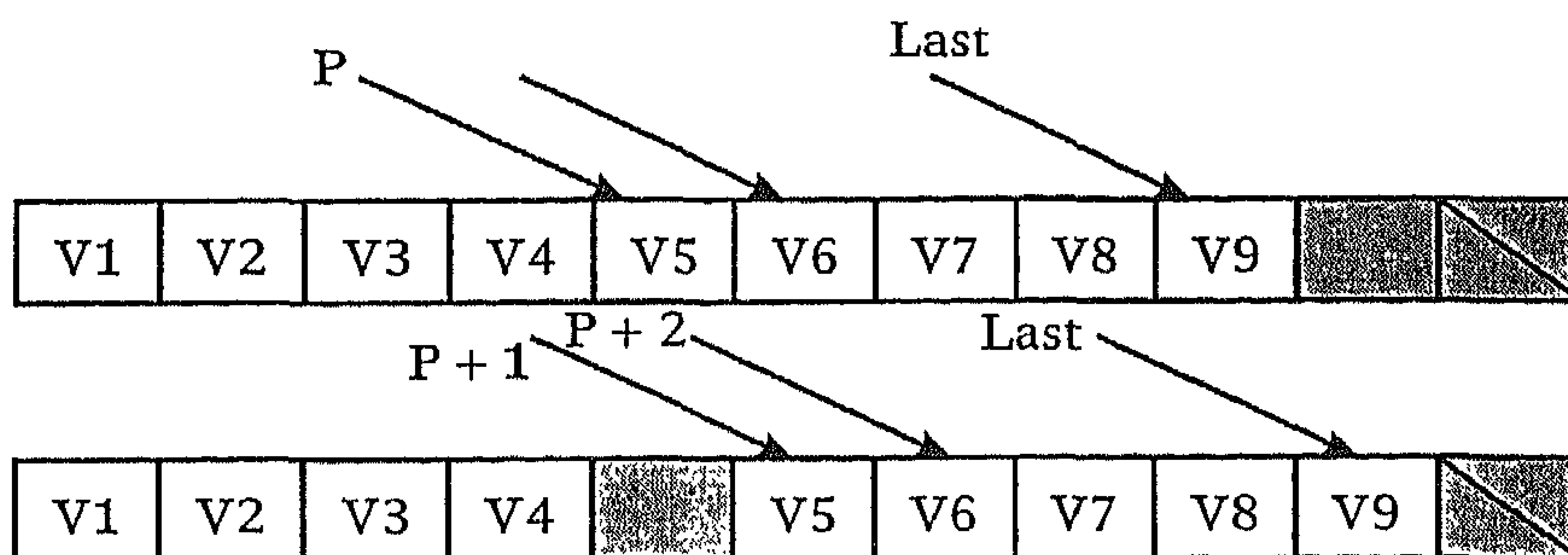
لقد لاحظنا فيما سبق أن عملية الإضافة بالنسبة للقوائم المتصلة التي تستند إلى استخدام المؤشرات تنطوي على حجز لمكان جديد في الذاكرة من أجل إضافة القيمة المعنية إلى القائمة. أما في حالة المصفوفات فإن الأمر مختلف من هذه الناحية، فالمكان محجوز منذ البداية، وبذلك فإن عملية الإضافة تقتصر فقط على إدخال القيمة في موضع محدد ضمن الحيز المتوفر في المصفوفة لأغراض الإضافة. وعلى خلاف القوائم المتصلة، فإننا قد نصل في حالة المصفوفات الأحادية إلى وضع لا نستطيع معه إضافة قيم جديدة إذا لم يتوفر مكان شاغر. وهذا الوضع هو ما نسميه الفائض (overflow).

لقد أشرنا من قبل في بداية هذه الوحدة إلى أن عملية الإضافة تنطوي على إزاحة للقيم الموجودة في المواضع التالية لموضع الإضافة وهي (p, p+1, ..., last) إلى المواضع التالية وهي (p+1, p+2, ... last+1) وذلك على النحو الممثل في الشكل (21). وينتج عن ذلك تغير قيمة المؤشر (last)، إذ تصبح (last+1) كما هو واضح. ومن هنا، يمكن أن نقول بأن عملية الإضافة تنطوي على ثلاث خطوات رئيسة هي:

أولاً: التأكد من وجود مكان شاغر في المصفوفة وتحديد مكان إضافة القيمة الجديدة ضمن مجموعة القيم المخزنة في المصفوفة.

ثانياً: إزاحة القيم التالية لمكان الإضافة إلى مواضع جديدة من أجل إخلاء موقع الإضافة في حالة وجود مثل هذه القيم.

ثالثاً: إدخال القيمة الجديدة في الموضع الذي تم إخلاؤه.



الشكل (21): عملية الإزاحة الناتجة عن إضافة قيمة جديدة للمصفوفة

ويمكن التعبير عن هذه الخطوات بالخوارزميات الثلاثة التالية:

الخوارزمية (7):

```
void MtList::insert (MtList& list ,  int element)
 {// inserting a new value into an array-based sorted list
  int p;
  if (list.last ==maxlength)
     cout<<»overflow»;
   else
    { p= 0;
     locatep(list,p, element);
     addelement(list,p, element);
     }
 } // end insert
```

الخوارزمية (8):

```
void MtList::locatep (MtList list, int& p, int element)
 {// finding the position for inserting a new value
  int i;
  bool found;
  if (list.last== 0)
    p= 0;   // first item
   else
    {i=0;
     found= false;
     while (!found && i <= last)
      if ( element > list.elements [i]) i++;
       else  found = true;
      p= i;
    }  } // end locatep
```

الخوارزمية (9):

```
void MtList::addelement (MtList& list ,int  p, int  element)
    {//shifting items down, inserting value, and updating value, and
updating last
    for (int  i=list.last; i>=p;i--) //shifting items
       list.elements [i+1]=list.elements[i];
     list.last++;  // updating pointer
     list.elements[p]= element;
    }//  addelement
```

وكما تلاحظ من هذه الخوارزميات الثلاثة، أننا افترضنا أن القيم التي تحويها المصفوفة منظمة وفق سـياق محدد. وفي حالة إضافة القيمة الجديدة إلى الموضع الأول

فإن الخوارزمية (6) يمكن أن تفعل ذلك بمفردها، أي دون الاستعانة بخوارزميات أخرى، أما في حالة الإضافة إلى نهاية القائمة فإن الأمر لا يحتاج لأكثر من إجراء التغيير على قيمة المؤشر (last) (إذا توفر الحيز الكافي في المصفوفة)، ومن ثم إسناد القيمة إلى الموضع المشار إليه بواسطة هذا المتغير.

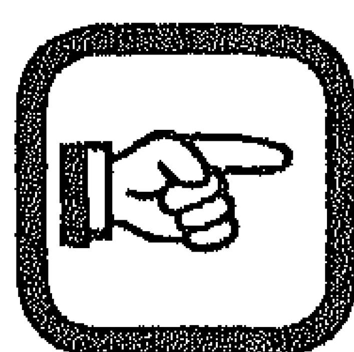
وإذا تأملت هذه الخوارزميات جيداً، فإنك ستلاحظ أن مسألة تحديد مكان الإضافة محكومة بثلاثة أوضاع هي:

أولاً: إن القائمة مليئة، وبذلك فإن المتغير (last) يشير إلى آخر موضع في المصفوفة، وبالتالي فإن لدينا وضع الفائض (overflow) الذي سبق أن أشرنا إليه.

ثانياً: إن القائمة خالية، وبذلك فإن القيمة الجديدة ستكون العنصر الأول والوحيد في القائمة.

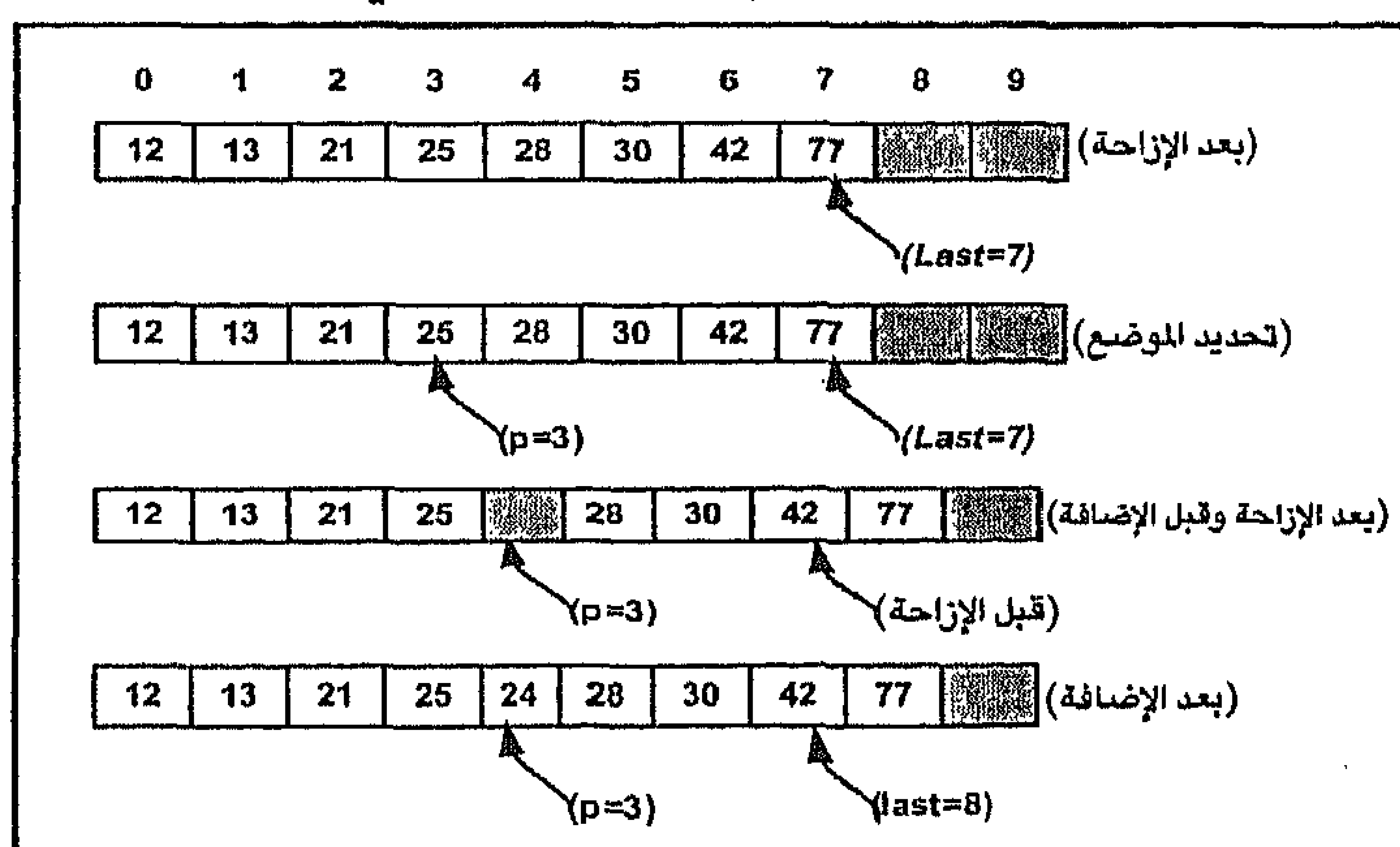
ثالثاً: إن في القائمة عنصراً واحداً على الأقل، وبذلك فإن موضع إضافة القيمة الجديدة هو بداية القائمة أو خلالها أو آخرها حسب ترتيبها ضمن مجموعة القيم التي تضمنها القائمة.

ولكي نوضح الأفكار والخطوات التي انطوت عليها هذه الخوارزميات، دعنا نورد المثال التالي:



مثال (6)

افترض أن لدينا قائمة مكونة من ثمانِ قيم في مصفوفة نرمز إليها بالاسم list ونرغب في إضافة القيمة (24) إليها، فإن ذلك سيتم على النحو التالي (انظر الشكل (22)):



الشكل (22): إضافة القيمة 24 إلى القائمة list

أولاً: يتم استدعاء الخوارزمية (7) ونرسل إليها البيانات المطلوبة لإجراء عملية الإضافة وهي: المصفوفة list والعنصر المراد إضافته (element).

ثانياً: يتم تنفيذ الجزء الأول من الجملة الشرطية المركبة التي تضمها الخوارزمية (7) if (last==maxlength) والإجابة هي بالنفي. ومن ثم يتم تنفيذ البديل else وبالتالي تصبح القيمة الابتدائية للمتغير p صفراً، ثم يجري البحث عن موضع لإضافة القيمة الجديدة وذلك من خلال استدعاء الخوارزمية (7) locatep.

ثالثاً: يبدأ تنفيذ الدالة (locatep) وتتسلسل خطوات التنفيذ على النحو التالي:

أ. ؟ last == 0 والإجابة بالنفي ولذلك يتم تنفيذ البديل else .

ب. found = false, i = 0 إعطاء قيم ابتدائية.

ج. ؟(!found && i <= last) الإجابة بالإثبات.

د. element > 12 والإجابة بالإثبات.

هـ. i = 1 تزداد قيمة المتغير i بمعدل واحد.

و. يتكرر اختبار شرط الدوران مرة أخرى على النحو المذكور في الخطوة (ج) أعلاه والإجابة بالإثبات.

ز. ؟ element = 13 والإجابة بالإثبات.

ح. i= 2 ينفذ الدوران للمرة الثانية وقيمة i تصبح ثلاثة.

ط. إجابة شرطي الدوران هي بالإثبات للمرة الثالثة.

ي. ؟ element > 21 والإجابة بالإثبات.

ك. i = 3 .

ل. نتيجة اختبار الدوران هي بالإثبات للمرة الرابعة.

م. ؟ element > 25 والإجابة هذه المرّة هي بالنفي.

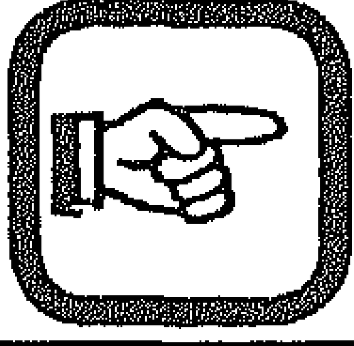
ن. found= true ومعنى ذلك أننا وجدنا موضع الإضافة.

س. نتيجة اختبار الدوران هي بالنفي هذه المرة لأن قيمة found أصبحت true.

ع. p= 3 تسند قيمة المتغير i إلى p (موضع الإضافة).

رابعاً: لقد عرفنا الآن الموضع الذي ستتم فيه الإضافة وهو list[3] والخطوة التالية هي استدعاء الخوارزمية (9). وفي هذه الخوارزمية تتم إزاحة العناصر بمعدل خانة واحدة باتجاه نهاية القائمة ابتداء من القيمة الأخيرة وانتهاء بموضع الإضافة، وبمجرد استكمال عملية الإزاحة، يتم إسناد القيمة المضافة إلى موضعها ونصل إلى الوضع الممثل في الجزء الأخير من الشكل (22).

تدريب (5)



افترض أننا نرغب في إضافة القيمة (80) إلى القائمة الأصلية الموضحة في الشكل (22)، فبين:

أ. عدد مرات تنفيذ جملة الشرط if المضمنة في التركيب الدوراني (while) الخاص بالدالة locatep.

ب. قيمة المتغير i بعد الانتهاء من تنفيذ الدوران في (locatep).

ج. عدد مرات تنفيذ التركيب الدوراني (for) الموجود في الدالة addelement.

ثم افترض أيضاً أننا نرغب في إضافة القيمة (10) هذه المرة إلى القائمة الأصلية في الشكل نفسه، وأجب على النقاط الثلاثة المذكورة أعلاه مرة أخرى لهذه الغاية.

ولعلك تلاحظ، عزيزي الدارس، من تدريب (5) ومثال (6) أن عدد القيم التي تحتاج إلى إزاحة باتجاه نهاية المصفوفة يعتمد على الموقع الذي ستحتله القيمة الجديدة. فكلما اقتربنا من بداية المصفوفة زاد عدد العناصر التي تحتاج إلى إزاحة. وعلى العكس من ذلك، كلما اقترب موضع الإضافة من نهاية المصفوفة، قل عدد القيم التي تحتاج إلى إزاحة. ومعنى ذلك أن عدد القيم المزاحة يتراوح بين صفر (وهي حالة الإضافة إلى نهاية القائمة) والعدد الكلي للقيم المخزنة (n). وبصفة عامة، فإن متوسط عدد القيم المزاحة هو حوالي نصف عدد القيم تقريباً. ويمكن أن يحتسب ذلك على أساس العلاقة التالية:

$$M_i = \sum_{k=1}^{n} (n-k+1).p_k$$

حيث: M_i: يشير إلى المتوسط.

K : موضع الإضافة.

n : عدد القيم في القائمة.

P : احتمالية إضافة قيمة إلى الموضع K وتساوي (n/1)

$$= {}^{1}\!/_{n}\left[\sum_{k=1}^{n}(n+1) - \sum_{k=1}^{n} k\right]$$

$$= (n(n+1)/n) - 1/n(\, n(n+1)/2)$$

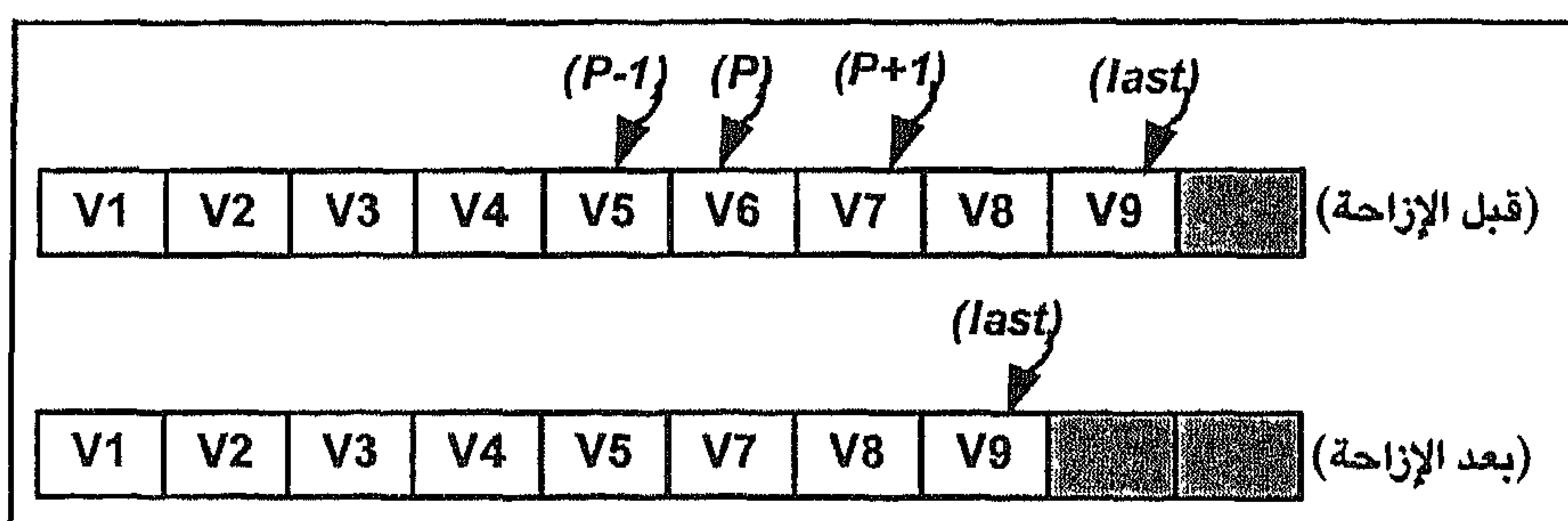
$$= n+1-(n+1)/2$$

$$= (n+1)/2$$

ج. حذف القيم (deleting values) من المصفوفة الأحادية

كما هو الحال بالنسبة لعملية الإضافة، فإن عملية الحذف تنطوي على إلغاء القيمة وليس الموضع. فالحجز للمصفوفة يبقى طيلة مدة تنفيذ البرنامج الذي عُرّفت فيه، رئيساً كان أو فرعياً. وهذا على خلاف القوائم المتصلة، التي تنطوي عملية الحذف فيها على إلغاء للموضع والقيمة على السواء.

وكما أشرنا في بداية هذه الوحدة فإن عملية الحذف تستلزم القيام بتحريك القيم التالية لموضع الحذف باتجاه بداية القائمة لسد الفجوة الناتجة عن عملية الحذف، فإذا تم الحذف في الموضع (p) فإن القيمة المخزنة في الموضع التالي لهذا الموضع (p+1) ستزاح إلى الموضع (p) حيث جرى الحذف، وقيمة الموضوع (p+2) ستزاح إلى الموضع (p+1)،... وهكذا حتى نصل إلى القيمة الأخيرة التي تزاح إلى الموضع (last-1) وذلك على النحو الموضح في الشكل (23). وينتج عن ذلك تغيير قيمة المتغير (last) بحيث تصبح (last-1).



الشكل (23): عملية الإزاحة الناتجة عن حذف قيمة من المصفوفة

ومن هنا، يمكن القول بأن عملية الحذف تنطوي على خطوتين أساسيتين هما:

أولاً: التأكد من وجود عناصر في المصفوفة وتحديد مكان القيمة المراد حذفها.

ثانياً: إزاحة القيم، إذا اقتضى الأمر ذلك، إلى بداية القائمة لسد الثغرة الناتجة عن الحذف.

ويمكن التعبير عن هاتين الخطوتين بالخوارزميات الثلاثة التالية (10-12):

الخوارزمية (10):

```
void MtList::delet(MtList& list, int element)
 {// deleting a value from an array-based dense list
  int p;
  if (list.last== -1) // list empty
```

```
cout<<»underflow»;
else
 {p= -1;
 findp(list, p, element);
 if (p== -1) cout<<»element is not in list»;
 else
 deleteval(list, p);
 }
} //  delete
```

الخوارزمية (11):

```
void MtList::findp (MtList list,int& p, int element)
 {// finding the position of the value to be deleted
  int i=0;
  bool found=false;
 while (!found && i < list.last)
   if ( element== list.elements [i])
    found = true;
   else  i++;
  if ( found == true)
   p=i;
   else
    p= -1;
 } // end  findp
```

الخوارزمية (12):

```
void MtList::deleteval(MtList&  list ,int  p)
 {//  deleting the value from the array at position p
  for (int  i=p; i<list.last;i++) //shift values up
   list.elements[i]=list.elements [ i+1];
   list.last--;
 }//end deleteval
```

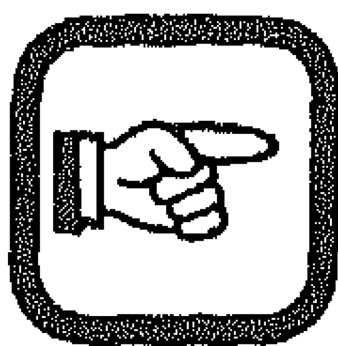
ولعلك تلاحظ، عزيزي الدارس، إننا نستطيع تطبيق هذه الخوارزميات الثلاثة على المصفوفات التي تتضمن قيماً لا يحكمها أي نظام ترتيبي محدد. وإذا تأملت هذه الخوارزميات جيداً، ستجد أن مسألة تحديد موقع القيمة المراد حذفها تنطوي على ثلاثة أوضاع مختلفة هي:

أولاً: أن تكون المصفوفة خالية، وبالتالي فإن القيمة التي نبحث عنها غير موجودة.

ثانياً: أن تكون القيمة غير موجودة ضمن قائمة القيم المخزنة في المصفوفة.

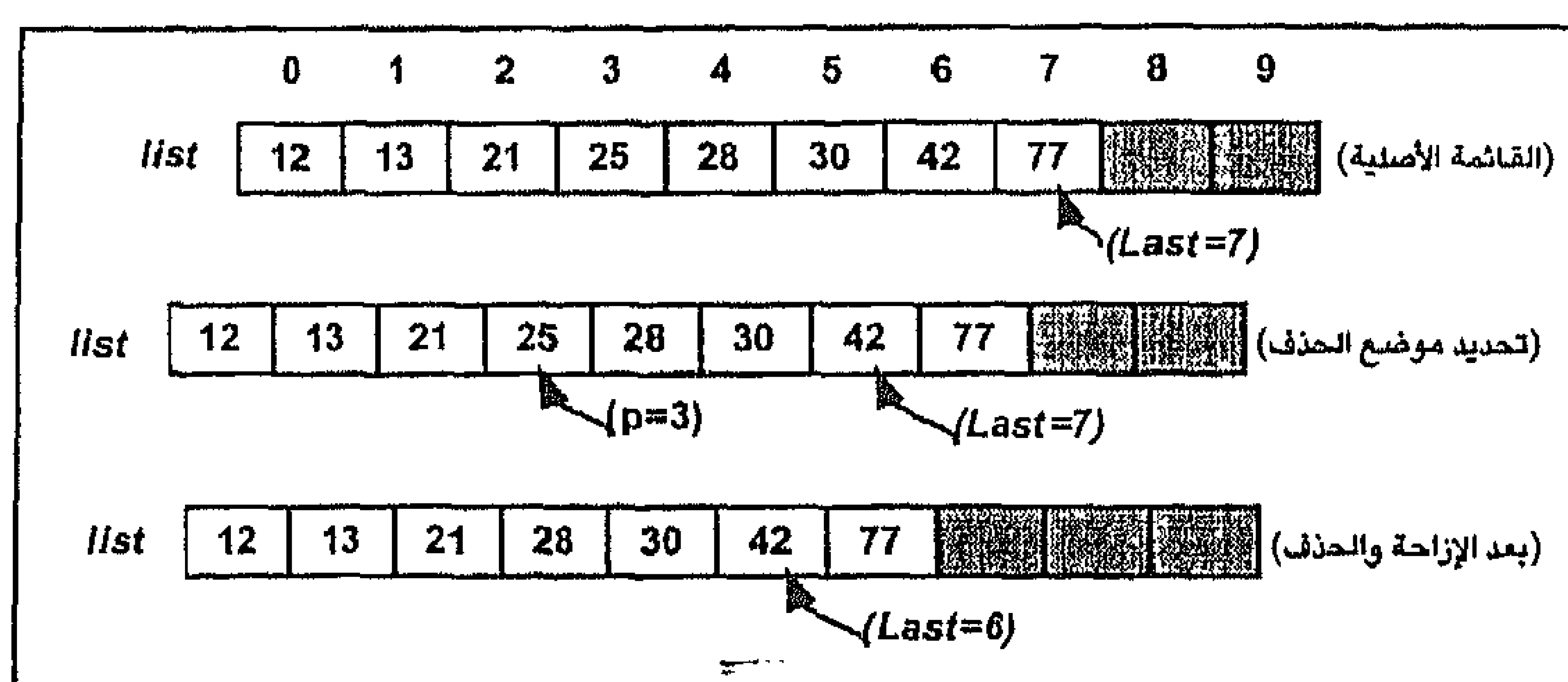
ثالثاً: أن تكون القيمة موجودة في مكان ما من القائمة، وقد يكون هذا المكان بدايتها أو نهايتها أو خلالها.

ولكي نوضح الأفكار والخطوات التي انطوت عليها هذه الخوارزميات الثلاث دعنا نورد مثالاً على ذلك.



مثال (7)

لنعد مرة أخرى إلى القائمة التي تعاملنا معها في عملية الإضافة، ولنر ما يحدث لو أردنا القيام بحذف القيمة (25) من القائمة (انظر الشكل (24)):



الشكل (24): حذف إحدى القيم من المصفوفة

أولاً: يتم استدعاء الخوارزمية (10) (delete) مع تزويدها بالبيانات التالية:

- القائمة (list) التي سيتم عليها الحذف.
- القيمة (element) التي سيتم حذفها.

ويتم تنفيذ الجملة الشرطية (last = 0 ?) وتكون النتيجة بالنفي وبذلك ننتقل إلى البديل else حيث يتم استدعاء الخوارزمية (11) (findp) وتزويدها بالبيانات اللازمة.

ثانياً: بعد إعطاء القيم الابتدائية للمتغيرين (i) و(found) في الدالة (findp) يجري تنفيذ التركيب الدوراني على النحو التالي:

أ. يتحقق الشرط الدوراني، وبذلك ندخل إلى الجملة الشرطية في الداخل:

element = 12 ? والإجابة هي بالنفي.

إذن: i = 1

ب. يتحقق الشرط الدوراني للمرة الثانية ويتم تنفيذ جملة if الداخلية:

? element = 13 والإجابة هي بالنفي.

إذن: i = 2

ج. يتحقق الشرط الدوراني ويتم تنفيذ جملة if الداخلية للمرة الثالثة:

? element = 21 والإجابة هي بالنفي.

إذن: i = 3

د. يتحقق الشرط الدوراني وتنفذ جملة if الداخلية للمرة الرابعة:

? element = 25 والإجابة هذه المرة هي بالإثبات.

إذن: found= true

هـ. لا يتحقق الشرط الدوراني في المرة الخامسة وبذلك يتم الانتقال إلى الجملة التالية، حيث يتم السؤال عن قيمة found:

? found = true والإجابة هي بالإثبات.

إذن: p= 3 وهو موضع الحذف (كما هو واضح في الشكل (24)).

ثالثاً: بعد استكمال تنفيذ الدالة (findp) نعود إلى الخوارزمية (10) نسأل عن قيمة (p) هكذا: (? p = -1) والإجابة هي بالنفي ومن هنا، يتم استدعاء الخوارزمية (12) حيث يجري ما يلي:

أ. يتم تحريك كل قيمة تالية للقيمة المحذوفة خانة واحدة باتجاه البداية وذلك على النحو الموضح في الشكل (24).

ب. يتم تغيير قيمة المتغير (last) بحيث تصبح (last-1).

وبانتهاء تنفيذ هذه الخوارزمية تنتهي عملية الحذف بأكملها.



تدريب (6)

افترض أننا نرغب في حذف القيمة (77) من القائمة الأصلية (list) الموضحة في الشكل (24)، فبين:

أ. عدد مرات تنفيذ جملة (if) المضمنة في التركيب الدوراني (while) الخاص بالدالة (findp).

ب. قيمة المتغير (i) بعد الانتهاء من تنفيذ التركيب الدوراني في الدالة (findp).

ج. عدد مرات تنفيذ التركيب الدوراني (for) الموجود في الدالة (deleteval).

وافترض أيضاً أننا نرغب في حذف القيمة (12) من القائمة الأصلية نفسها (list) الموضحة في الشكل (24)، فأجب على النقاط المذكورة أعلاه مرة أخرى، ثم لاحظ الفرق بين الإجابتين.

ولعلك تلاحظ، عزيزي الدارس، من تدريب (6) ومثال (7) أن عدد القيم التي تحتاج إلى إزاحة باتجاه بداية المصفوفة يعتمد على الموقع الذي تحتله القيمة المراد حذفها. وكما هو الشأن بالنسبة لعملية الإضافة، فإن عدد القيم المزاحة يزيد كلما اقتربنا من البداية. وهذا العدد يتراوح بين الصفر والقيمة **n - 1** (حيث الرمز n يشير إلى العدد الكلي للقيم). وفي المتوسط فإن عدد القيم المزاحة يصل إلى حوالي النصف. ويمكن احتساب ذلك على أساس المعادلة التالية:

$$M_d = \sum_{k=1}^{n} (n-k)\; p_k$$

حيث أن:

M_d يشير إلى عدد القيم المزاحة.

K يشير إلى موضع الحذف.

n عدد القيم في القائمة.

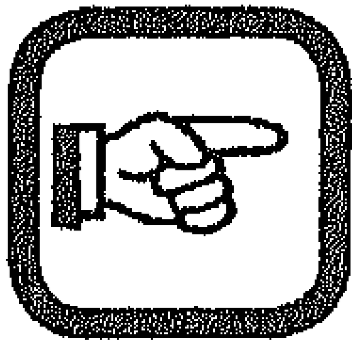
P احتمالية حذف قيمة من الموضع k وتساوي **1** / n.

$$
\begin{aligned}
&= \sum_{k=1}^{n} (n-k)\; p_k \\
&= 1/n \sum_{k=1}^{n} n - \frac{1}{n} \sum_{k=1}^{n} k \\
&= n - (n(n+1)/2n \\
&= n - (\; n+1)/2) \\
&= (n-1)/2
\end{aligned}
$$

4. محاكاة القوائم المتصلة باستخدام المصفوفات

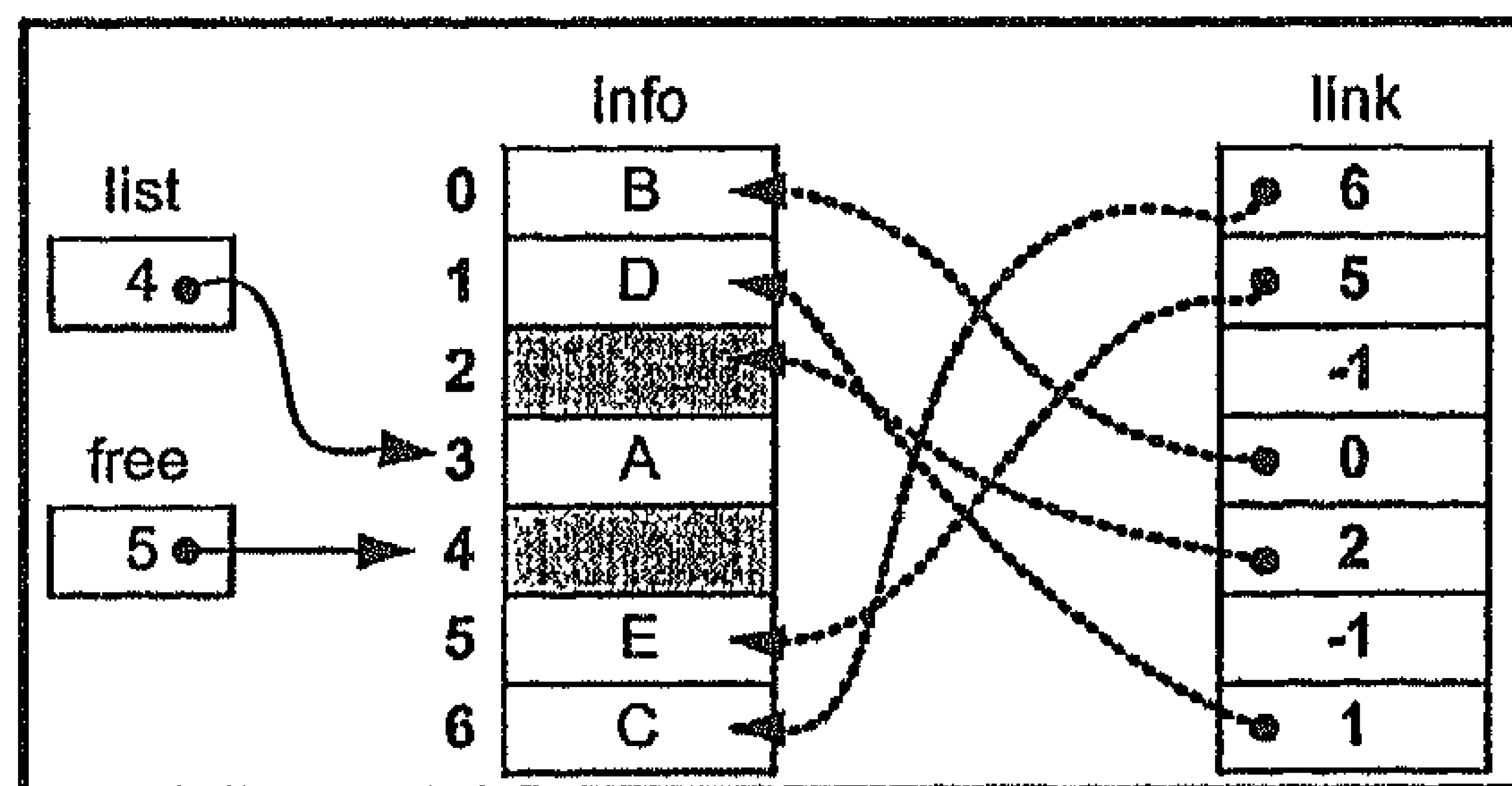
كما أشرنا في الوحدة الأولى من هذا المقرر، عزيزي الدارس، فإن هذا الأسلوب الموضح أعلاه في تمثيل القوائم المتصلة هو ما نصطلح عليه باسم التمثيل المتصل باستخدام المؤشرات (pointer implementation). وهو يصلح بصفة خاصة في اللغات التي توفر للمبرمج إمكانات برمجية تسمح له بتعريف المؤشرات والإفادة منها في الربط بين العناصر المتتابعة.

وبالإضافة إلى ذلك، هناك طريقة أخرى لتنفيذ القوائم المتصلة تقوم على استخدام المصفوفات المتوازية وفكرة الدالة النسبية (cursor-based implementation) أي نسبة إلى بداية المصفوفة. وهذه الطريقة تصلح بصفة خاصة في اللغات التي لا توفر نمطاً إشارياً متميزاً مثل فورتران والجول. كما يمكن تطبيقها في اللغات التي توفر مثل هذا النمط، إلا أن استخدام أسلوب المؤشرات يمتاز عليه في كثير من الجوانب. ولكي نوضح مفهوم هذا النوع من التمثيل، دعنا نتأمل المثال التالي.



مثال (8)

افترض أن لدينا خمس قيم رمزية هي: (A, B, C, D, E)، وهي القيم نفسها التي تعاملنا معها في المثال السابق، ونريد أن نمثلها على شكل قائمة متصلة، ولكن هذه المرة باستخدام الدالات النسبية والمصفوفات المتوازية. فكما هو الحال بالنسبة لطريقة تمثيل القوائم المتصلة باستخدام المؤشرات، يتكون كل عنصر من جزئين: الأول هو القيمة المعنية، والآخر هو الإشارة الدالة على العنصر التالي. ومعنى ذلك أننا سنحتاج إلى مصفوفتين متوازيتين: الأولى لتخزين القيم والأخرى لتخزين الدالات النسبية (cursors) التي تصل العناصر ببعضها. والشكل (25) يوضح طريقة تمثيل القائمة باستخدام هذا الأسلوب.



الشكل (25): تمثيل القوائم المتصلة باستخدام المصفوفات المتوازية والدالات النسبية

ولو حاولنا أن نعبر عن هذه الصيغة في التمثيل، لكان لدينا التعريف التالي:

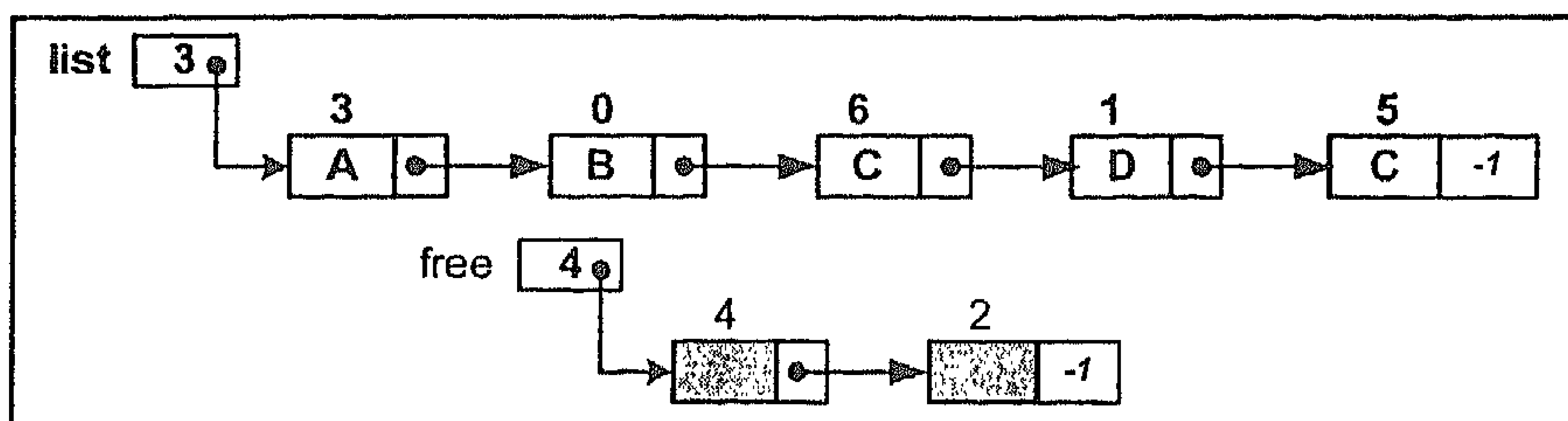
```
char info[7];
int link[7];
int list,free;
```

وكما هو واضح من الشكل (25) والتعريف المصاحب له، فإن كلا المصفوفتين من حجم واحد. وبذلك، جرياً على مفهوم المصفوفات المتوازية، فإن كشاف (index) كل منهما مماثل للآخر، وكذلك الحال بالنسبة للشارة الدالة على موضع العنصر (subscript). فإذا قلنا: info[0] فإننا نعني بذلك العنصر الأول في المصفوفة link أيضاً. ولعلك تلاحظ أيضاً من الشكل (25) ومن التعريف البرمجي أن هناك متغيرين آخرين: الأول هو list ومهمته هي الإشارة إلى العنصر الأول في القائمة المتصلة، والثاني هو free ومهمته هي الإشارة إلي العنصر الأول في قائمة الأماكن الخالية في المصفوفة info . والقيمة المخزنة في كل منهما هي عدد صحيح، وبحكم طبيعة المهمة الخاصة لكل منهما فإن القيمة لا تكون غير ذلك.

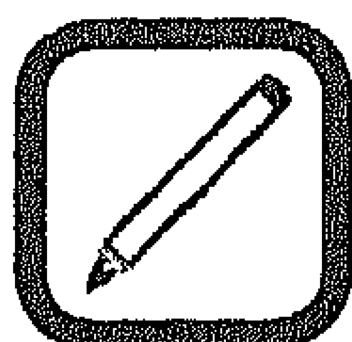
أما فيما يتصل بالقيم الواردة في المصفوفتين، فمن الواضح أن القيم المخزنة في info هي تلك القيم الرمزية التي تشكل عناصر القائمة المتصلة. وقد تعمدنا وضع هذه القيم في أماكن متفرقة داخل المصفوفة لبيان أن عناصر القائمة لا تأتي بالضرورة متجاورة في هذه المصفوفة، بل تخزن حيثما كان هناك مكان شاغر. ويتم الإحتفاظ بالعلاقة المنطقية القائمة بين هذه العناصر من خلال القيم المذكورة في المصفوفة link. فكما هو واضح من اتجاه الأسهم في هذا الشكل، هذه القيم تمثل الدالات النسبية إلى العناصر التالية. ومن هنا نستطيع تتبع تسلسل هذه الدالات على النحو التالي (انظر تسلسل اتجاهات الأسهم):

list = 3	وبذلك فإن	info [3] ='A'
link [3] = 0	وبذلك فإن	info [0] ='B'
link [0] = 6	وبذلك فإن	info [6] ='C'
link [6] = 1	وبذلك فإن	info [1] = 'C'
link [1] = 5	وبذلك فإن	info [5] = 'E'
link [5] = -1	وبذلك فإن	هذه نهاية القائمة

ويمكن التعبير عن هذا التسلسل الخاص، وتسلسل الأماكن الخالية على النحو الممثل في الشكل (26).



الشكل (26): التسلسل المنطقي للقيم والمواضع الخالية في الشكل (25)



تدريب (7)

افترض أن لدينا عدداً من الأطباء في أحد المستشفيات ونريد الاحتفاظ بالمعلومات التالية عن كل منهم: إسم الطبيب، وجنسه (ذكراً أم أنثى)، وعمره، وراتبه. وافترض أيضاً أننا نريد أن نحتفظ بالمعلومات نفسها عن الممرضات اللواتي يعملن في المستشفى. فإن مثل هذه المعلومات يمكن أن تمثل بأسلوب المصفوفات المتوازية والدالة النسبية على النحو الموضح في الشكل (27). والمطلوب هو:

أن تقوم بتتبع العلاقة القائمة بين الأطباء الممثلين في هذا الشكل وتوضح بالرسم هذه العلاقة على شكل قائمة متصلة بالأسلوب نفسه المبين في الشكل (26). ثم تفعل الشيء نفسه بالنسبة لعلاقة الممرضات ببعضهن، ومن ثم علاقة الأماكن الخالية ببعضها.

أن تقوم بتعريف هذا التركيب البياني تعريفاً برمجياً بالأسلوب نفسه الذي بيناه في المثال السابق، وذلك باستخدام المصفوفات في لغة سي++.

أن تقوم بتعريف هذا التركيب البياني تعريفاً برمجياً مرة أخرى بالأسلوب نفسه الذي بيناه في المثال (1) وذلك باستخدام المؤشرات في سي++.

وبعد أن تنجز ذلك، قارن إجابتك بالإجابات المعطاة في نهاية هذه الوحدة.

free 2 → 2

Nurses 4 → 4

Doctor 7 → 7

	name	sex	age	sal	link
1	Faris	M	35	250	5
2					8
3	Jihad	M	29	240	0
4	Maha	F	24	150	10
5	Hind	F	29	240	3
6	Wisal	F	31	170	0
7	Anas	M	26	280	9
8					0
9	Bayan	F	28	260	1
10	Narjis	F	21	230	6

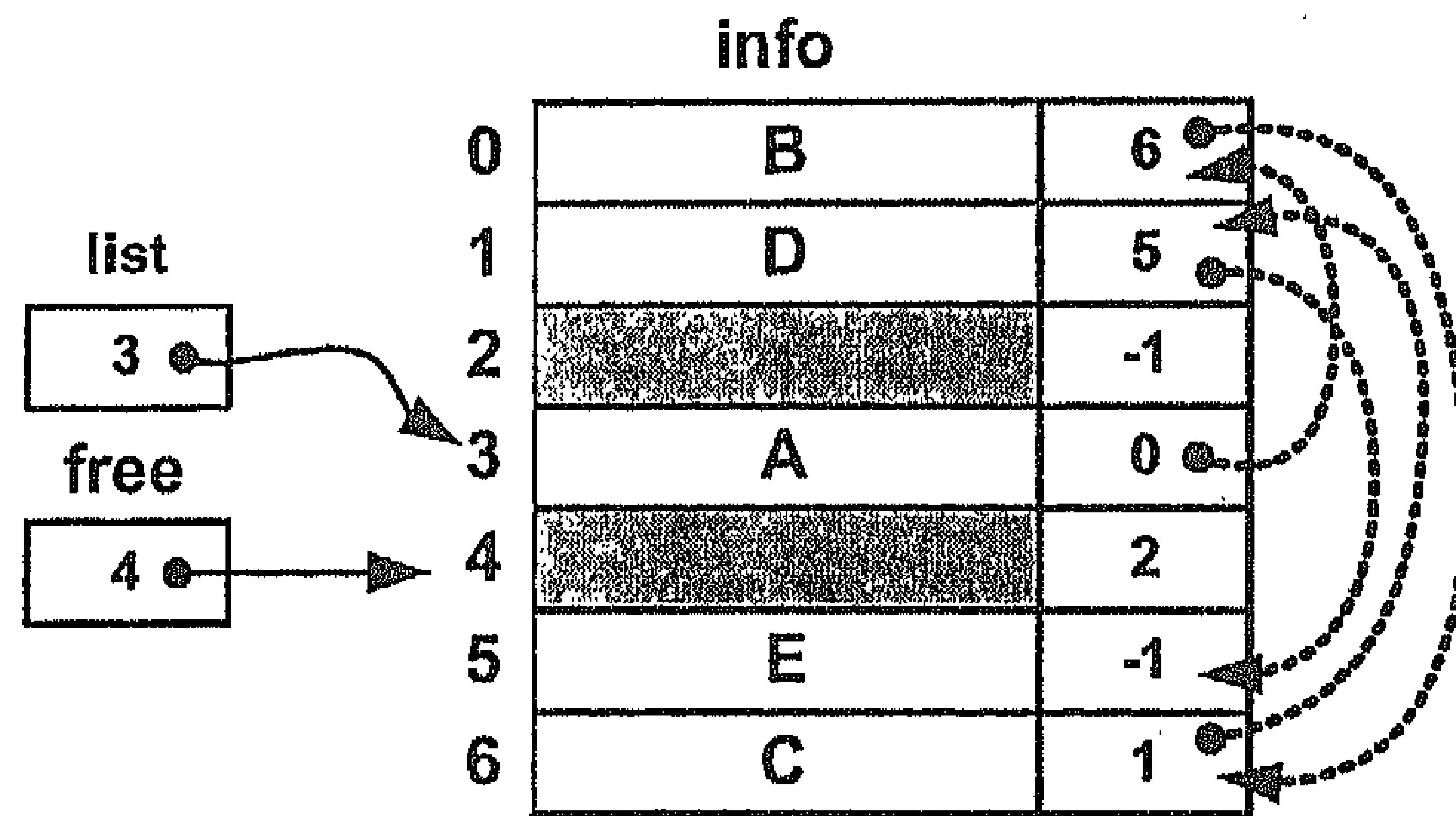
الشكل (27): أكثر من قائمة متصلة مخزنة في مكان واحد

وبالإضافة إلى الأسلوبين السابقين لتمثيل القوائم، فإن هناك أيضاً أسلوباً ثالثاً يقوم على استخدام المصفوفات المركبة من سجلات (composite arrays). أي أن كل عنصر في المصفوفة هو سجل مكون من عدد من الحقول، وأحد هذه الحقول يستغل لتخزين قيمة الدالة النسبية إلى العنصر التالي. ولتوضيح فكرة هذا النوع من التمثيل دعنا نعود مرة أخرى إلى مثالنا السابق ونعيد صياغة تمثيل القائمة بالأسلوب الجديد.

مثال (9)

للمرة الثالثة، نحن بصدد القيم الرمزية الخمسة التي استخدمناها في المثالين السابقين، وهي: (A, B, C, D, E). ونريد الآن أن نشكل منها قائمة متصلة باستخدام مصفوفة السجلات، وهو ما نعبر عنه على النحو الموضح في الشكل (28). ويمكن التعبير عن هذا النوع من التمثيل من الناحية البرمجية باستخدام التعريف التالي:

```
class aType
 { struct rec
    {char info;
     int link;
    };
rec arrayofrec[7];
} a;
int free, list;
```



الشكل (28): تمثيل القوائم باستخدام مصفوفات السجلات

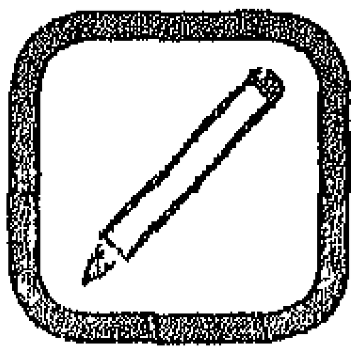
وكما هو واضح من هذا التعريف ومن الشكل (28)، أن هذا الأسلوب لا يختلف كثيراً عن أسلوب المصفوفات المتوازية. وما هذا الأسلوب، في الحقيقة، إلا صيغة معدلة لأسلوب المصفوفات المتوازية والدالات النسبية، وهو يصلح بصفة خاصة في اللغات التي تتضمن إمكانية تعريف السجلات وإمكانية تركيب المصفوفات من السجلات، على النحو المتوفر في لغة سي++.

وكما هو الحال بالنسبة لتمثيل القوائم المتصلة القائم على المصفوفات المتوازية، فإننا في هذه الصيغة ينبغي أن نقدر الحد الأعلى للقائمة ونقوم بتعريف مصفوفة كافية لهذا العدد. فالعدد في هذه الحالة يتم تثبيته خلال مرحلة الترجمة، ومن هنا يشار إلى المتغير A المذكور أعلاه بأنه متغير ثابت (static) طيلة مدة تنفيذ الجزء الذي ينتمي إليه. وعلى خلاف أسلوب التمثيل المتصل القائم على المؤشرات، فإن حجز العدد المناسب من المواضع للمتغير A يتم مرة واحدة، وقبل تخزين القيم وإيجاد القائمة. أما بالنسبة للأسلوب القائم على المؤشرات، فإن الحجز الوحيد الذي يتم خلال مرحلة الترجمة هو المتغيرlist أما بقية المواضع الخاصة بالقائمة فيتم حجزها وفق الحاجة، خلال مرحلة التنفيذ. وبذلك فإن الحجز يسير جنباً إلى جنب مع نمو القائمة نفسها. ومن هنا، فإننا نشير إلى المتغير الذي يستخدم في عملية الحجز بأنه متغير ديناميكي (dynamic).

والحقيقة أن مسألة الاختيار بين الأساليب الثلاثة السابقة يتوقف على عاملين أساسيين هما:

أ. مدى توفر إمكانية استخدام هذه الأساليب الثلاثة في لغة البرمجة المستعملة. فقد لاحظنا أن لغة الفورتران، مثلاً، ليس لديها إمكانات لتعريف المؤشرات ولا لتعريف السجلات. وبذلك فإن الوسيلة الوحيدة المتوفرة هي استخدام المصفوفات المتوازية. أما في لغة سي++، من جانب آخر، فإننا نجد جميع الإمكانات للتعامل مع الأساليب الثالثة المذكورة.

ب. مدى النمو المتوقع الذي تتسم به القائمة. فإذا تركنا مسألة السهولة في التعامل مع القائمة من حيث الإضافة والحذف جانباً، فإن الإحساس بوجود درجة عالية من التغير على القائمة يوجب اختيار أسلوب يتسم بالديناميكية، وهو ما يتوفر في التمثيل القائم على المؤشرات (في حالة وجود مثل هذه الإمكانية في اللغة). ونظراً للأهمية الخاصة للتمثيل المتصل القائم على المؤشرات، فإننا سنلتزم بهذا الأسلوب في الأجزاء المتبقية من هذه الوحدة.



تدريب (8)

استخدم المعلومات المعطاة في التدريب (7) لعمل ما يلي:

أ. تحويل الرسم المعطى إلى أسلوب تمثيل القوائم المتصلة باستخدام مصفوفات السجلات.

ب. تقديم تعريف بلغة سي++ لهذا التركيب الجديد بالأسلوب الذي وضحناه في المثال (9) أعلاه.

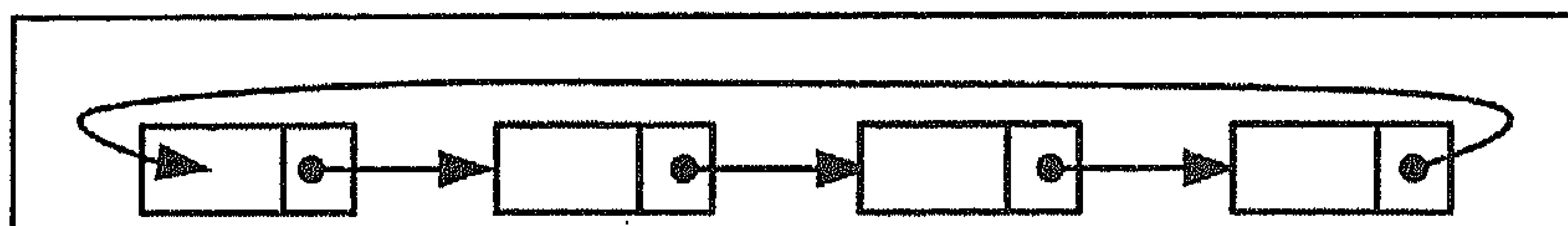
5. أنواع أخرى من القوائم المتصلة

في هذا القسم سنناقش، عزيزي الدارس، أشكالاً أخرى للقوائم المتصلة:

1.5 القوائم الدائرية وعملياتها (Circular Linked Lists)

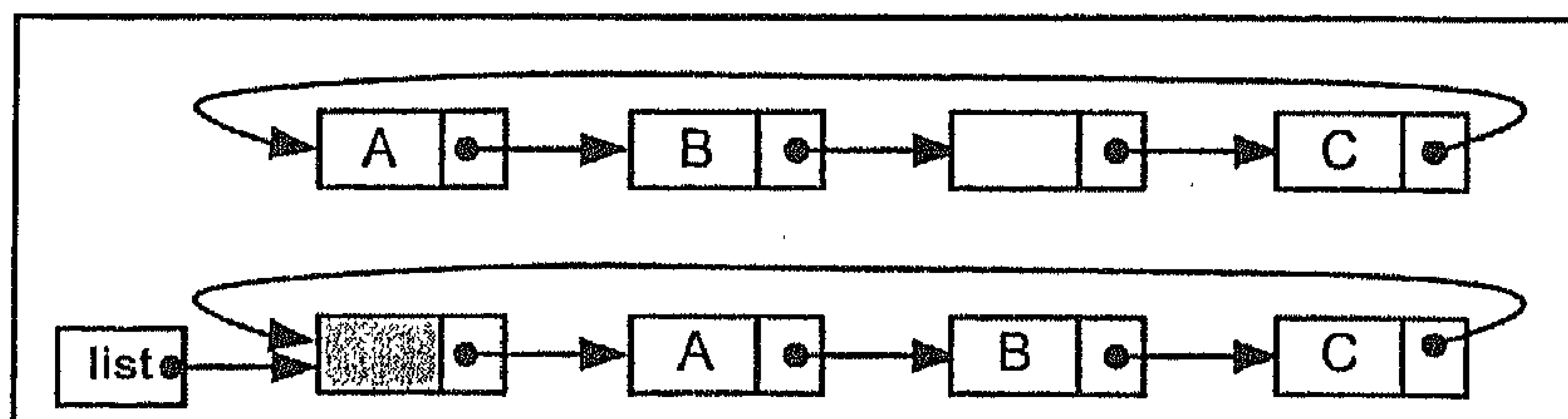
على الرغم من أن القائمة المفردة الخطية مهمة جداً في عمليات تركيب البيانات ومعالجتها، إلا أنها تعاني من بعض المشكلات. فإذا كنا، مثلاً، في نقطة معينة من القائمة فإنه بإمكاننا الوصول إلى العناصر التالية في القائمة، ولكنه ليس بإمكاننا أن نعود إلى أية عناصر سابقة لهذه النقطة دون العودة إلى بداية القائمة من خلال المؤشر الخارجي الخاص بها. ويكمن حل هذه المشكلة وغيرها من المشكلات في إضفاء ميزات جديدة على هذا التركيب البياني، بحيث نجعله أكثر فعالية. وهذا هو ما نجده في القوائم الدائرية والأشكال الأخرى للقوائم المتصلة التي سنشير إليها فيما بعد.

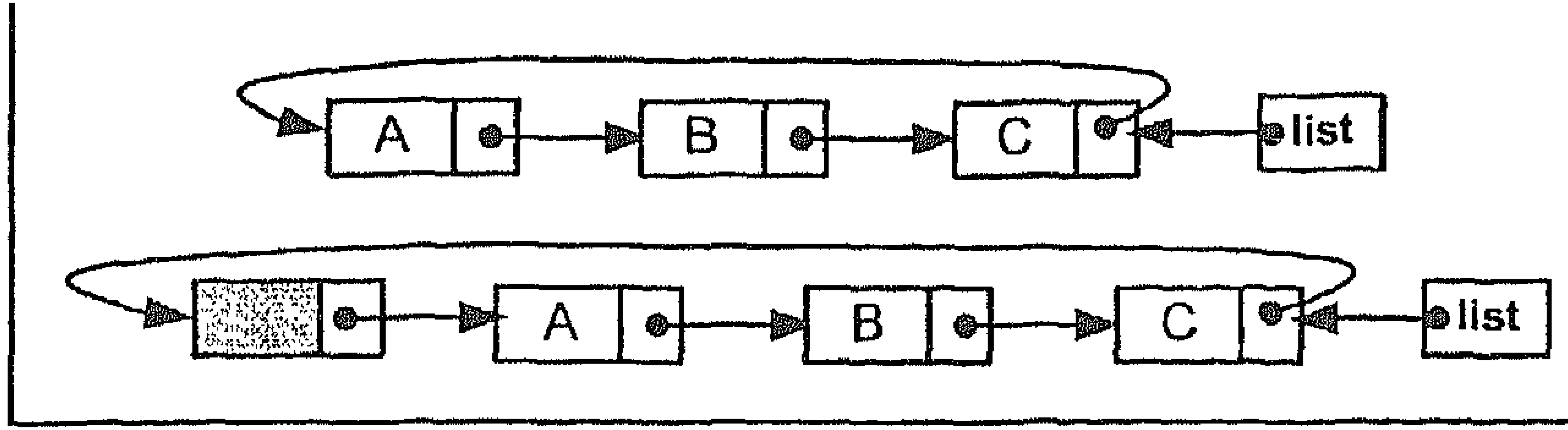
والميزة الإضافية الأساسية التي نجدها في القائمة الدائرية هي أننا نستطيع أن نعود إلى بداية القائمة، أو أي عنصر آخر فيها دون أن نضطر في كل مرة إلى أن نقوم بعملية الاستقصاء من خلال المؤشر الخارجي للقائمة. فلو تأملنا الشكل (29)، الذي يمثل إحدى صيغ القوائم الدائرية، لوجدنا أن بإمكاننا أن نستعرض عناصر هذه القائمة واحداً تلو الآخر ونعود بالنهاية إلى النقطة التي بدأنا منها.



الشكل (29): الصيغة العامة للقائمة الدائرية

ولكن، كما تلاحظ، فإن الصيغة الممثلة في هذا الشكل، وهي الصيغة الحقيقية المجردة للقائمة الدائرية، لا تنطوي على ما يشير إلى نقطة البداية أو النهاية، على النحو الذي شاهدناه في القائمة المفردة البسيطة. ومن هنا، فلا بد من الاتفاق على مثل هذه النقطة لتيسير عملية التعامل معها. وهناك، في الحقيقة، أكثر من وسيلة للوصول إلى ذلك مما يؤدي بالتالي إلى وجود صيغ متعددة للقوائم الدائرية نعرضها في الشكل (30).





الشكل (30): الصيغ المختلفة التي ترد بها القوائم الدائرية

فكما يتضح من هذا الشكل، فإن الصيغة الأولى للقائمة الدائرية مبنية على فكرة القائمة البسيطة نفسها من حيث وجود مؤشر خارجي إلى العنصر الأول (وهو list في هذا الشكل). والفارق الوحيد يتمثل بوجود رابطة بين العنصر الأخير والعنصر الأول في القائمة وذلك على نحو يجعلها دائرية الشكل. فقيمة مؤشر العنصر الأخير لم تعد NULL كما هو الحال في القائمة المفردة، بل أصبح يشير إلى بداية القائمة.

أما الصيغة الثانية للقائمة الدائرية، فإنها تشتمل على إضافة أخرى على الصيغة السابقة وهي وجود عنصر خاص في أول القائمة الدائرية. ولعلك تذكر أننا قد اصطلحنا على تسمية هذا العنصر باسم رأس القائمة أو مقدمتها. ولعلك تذكر أيضاً بأن الهدف من هذا الرأس هو الاحتفاظ ببعض المعلومات عن القائمة بالإضافة إلى تحديد البداية. ومن الممكن أيضاً في حالة وجود هذا الرأس أن نقوم بعملية استعراض للعناصر بمؤشر واحد، هو المؤشر الخارجي نفسه. وتستعمل هذه الصيغة في كثير من الأحيان بدلاً عن القوائم البسيطة نظراً لأنها تمتاز عليها بميزتين على الأقل، الأولى: هي أنه ليس هناك مؤشر يشتمل على قيمة NULL في التركيب البياني كله. والثانية: هي أن كل عنصر من العناصر الفعلية له سابق، وبذلك فإن العنصر الأول لا يحتاج إلى معاملة خاصة.

وأما الصيغتان الأخريان، الثالثة والرابعة، فإنهما يماثلان الصيغتين الأولى والثانية. والفرق هو في وجود المؤشر الخارجي (list) في نهاية القائمة وليس في بدايتها. والميزة الأساسية لهذا الاختلاف هي أننا في حالة وجود المؤشر إلى العنصر الأخير في القائمة، فإننا نستطيع أن "نضرب عصفورين بحجر واحد"، كما يقال. فبالإضافة إلى كونه يشير مباشرة إلى العنصر الأخير في القائمة، فإنه يستطيع أيضاً أن يشير إلى العنصر الأول في القائمة بطريقة غير مباشرة وذلك من خلال مؤشر العنصر الذي يشير إليه والذي يؤدي بنا- كما تلاحظ- إلى بداية القائمة. ولكن ينبغي أن نلاحظ بأن هذه الميزة ليست على درجة كبيرة من الأهمية، إلا في الحالات التي نرغب فيها بإضافة عناصر جديدة إلى نهاية القائمة كما هو الحال في الطوابير.

وفيما يتصل بالعمليات التي تتم على القوائم الدائرية، فإنها لا تكاد تختلف عن القوائم المفردة البسيطة إلا في المسائل المتعلقة بقيمة مؤشر العنصر الأخير في القائمة، كاختبار نهاية القائمة في حالة البحث عن موقع أحد العناصر أو استعراض العناصر، وكعملية حذف آخر عنصر في القائمة. وفيما يلي نستعرض أهم خوارزميات العمليات الخاصة بهذا النوع من القوائم المتصلة، وسنقصر حديثنا على الصيغة الثانية للقوائم الدائرية لما تنطوي عليه من ميزات.

1.1.5 بناء القوائم الدائرية (circular list creation)

إن عملية بناء القائمة الدائرية بالصيغة التي تم تبنيها هنا، تنطوي على خطوتين أساسيتين: الأولى تمهيدية يتم خلالها تكوين الرأس، والأخرى يتم خلالها إضافة العناصر الفعلية إلى القائمة الابتدائية. ومن هنا، فإن خوارزمية تكوين الرأس لا تُستدعى إلا مرة واحدة، بينما يتم تنفيذ خوارزمية بناء القائمة الفعلية عدداً من المرات مساوٍ لعدد عناصر القائمة. وفيما يلي نستعرض تفاصيل هاتين الخوارزميتين.

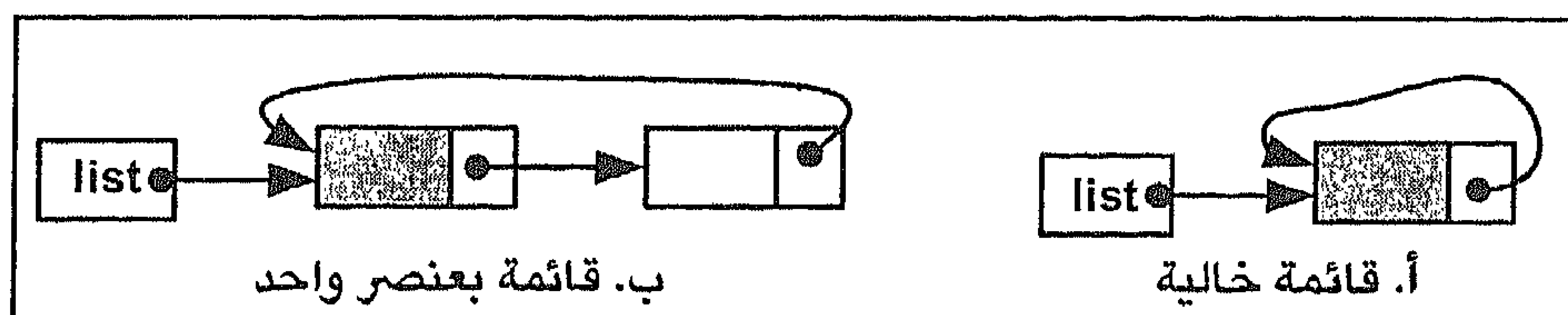
الخوارزمية (13):

```
crList* crList::creatHdr(crList* list,crList* last,int val)
 { crList *header= new crList; // create header node
   header->info=val;  // assign  a value to header
   header->link=header;  // header points to itself
   list=header;
   last=header; // LAST keeps track of end
 return list;
 }
```

الخوارزمية (14):

```
crList* crList::creatCl(crList* list,crList* last, int element)
{ crList *nextNode= new crList; // create header node
  nextNode->info=element;  // assign  a value to header
  last->link=nextNode;
  nextNode->link=list;
  last=nextNode;
  return last;
 }
```

ولو تأملنا هاتين الخوارزميتين فإننا سنجد أن الأولى (createhdr) تقوم بالخطوات التمهيدية لبناء القائمة. وعند تنفيذ هذه الخوارزمية فإن صورة القائمة ستكون على النحو الموضح في الجزء الأول من الشكل (31 -أ). حيث تلاحظ أن الرأس يشير إلى نفسه في هذه المرحلة، وذلك نتيجة لتنفيذ الجملة الثالثة في الخوارزمية. ولعلك تلاحظ أيضاً بأننا قد استعملنا مؤشراً خارجياً إضافياً هو (last) بهدف متابعة نهاية القائمة لتسهيل عملية الإضافة إلى نهاية القائمة الدائرية، وهو الأسلوب المعتمد في خوارزمية البناء (14).



الشكل (31): أسلوب بناء القائمة الدائرية.

أما بعد تنفيذ الخوارزمية الأخرى (createcl) وإضافة العنصر الأول فيها فإن الرأس يبدأ بالإشارة إلى هذا العنصر ويشير العنصر بدوره إلى الرأس للمحافظة على الطبيعة الدائرية للقائمة (على النحو الموضح في من الشكل (31 - ب) وذلك من خلال تنفيذ الجملة الرابعة في هذه الدالة. ولعلك لاحظت بأننا لم نحتج في هذه الخوارزمية إلى اختبار ما إذا كنا بصدد التعامل مع العنصر الأول في القائمة، كما هو الحال في الخوارزمية (1) (createl) التي مر ذكرها عندما قلنا:

```
if ( list==NULL)
list= nextnode
else          ...
```

كما أن العنصر الحقيقي الأول له سابق وهو الرأس وبالتالي فإن قيمة المؤشر last عند استدعاء (createcl) لأول مرة لا تكون NULL . وبذلك فقد استطعنا بهذه الصيغة تبسيط عملية بناء القائمة المتصلة.

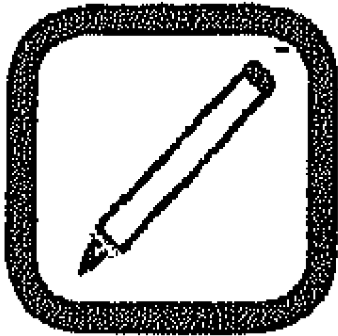
2.1.5 استعراض القوائم الدائرية (Circular List Traversal)

إن عملية استعراض القائمة الدائرية لا تختلف في جوهرها عن استعراض القائمة المفردة، إلا أن هناك فرقين أساسيين في أسلوب تطبيق العملية، الفرق الأول: هو أننا نبدأ بالعنصر التالي للرأس وبذلك فإن الجملة التالية في خوارزمية الاستعراض (1) المذكورة

سابقاً: (current= list) لا بد أن تعدل بحيث تصبح (current = list ->link) . والفرق الثاني يكمن في الطبيعة الدائرية للقائمة، حيث سننتهي إلى النقطة التي بدأنا منها. وبذلك فإن الاختبار الدوراني المتصل بنهاية القائمة لا بد وأن يأخذ بالاعتبار أن المؤشر سيشير في النهاية إلى الرأس. وفيما يلي تفاصيل عملية الاستعراض كما تعرضها الخوارزمية (15).

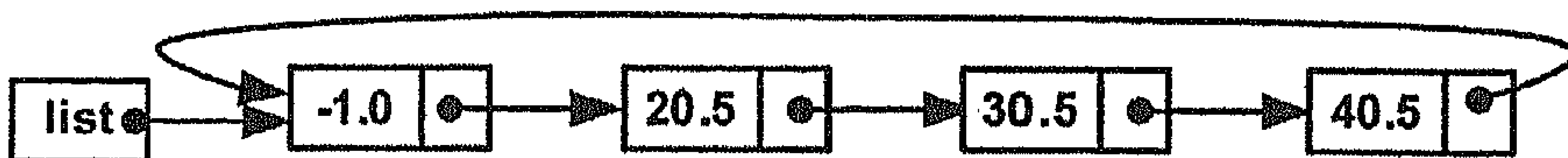
الخوارزمية (15):

```
void crList::traverseCl(crList * list)
  {crList *current;
   current=list->link;
   while (current!= list)
     {process(current->info);
      current=current->link;
     }// end while
  }
```



تدريب (9)

افترض أن لدينا قائمة دائرية ذات قيم حقيقية (real) موجبة تتراوح بين الصفر والمائة، وافترض أن القيمة المخزنة في رأس القائمة هي قيمة سالبة وذلك على النحو الموضح في الشكل (32). والمطلوب هو إجراء تعديل على الخوارزمية المذكورة أعلاه، بحيث نستخدم فيها المؤشر الخارجي (list) فقط لاستعراض العناصر بدلاً من المؤشر المحلي (current). ولعلك تلاحظ، بأننا استخدمنا رأس القائمة لتخزين قيمة خارج نطاق القيم الممثلة في القائمة الفعلية، وبذلك نستطيع أن نستخدم هذه القيمة للسؤال عن بداية القائمة بدلاً من تثبيت المؤشر الخارجي (list) واستخدام مؤشر آخر للتحرك عبر القائمة.



الشكل (32): قائمة دائرة ذات رأس له قيمة خاصة.

3.1.5 إضافة عناصر جديدة إلى القوائم الدائرية (Insertion)

نظراً لأن الحد الأدنى الذي ننطلق منه في إطار الصيغة الدائرية التي نحن بصدد التعامل معها هو وجود حالة مبدئية للقائمة متمثلة برأس القائمة، فإنه لم يعد من المجدي أن نسأل فيما إذا كانت القائمة خالية أو لا، ولا أن نسأل أيضاً فيما إذا كان العنصر هو الأول في القائمة. فالمطلوب هو تحديد موقع الإضافة وإنجاز هذه العملية بصرف النظر عن مسألة البداية أو النهاية أو الوسط التي شغلنا أنفسنا بها في القوائم المفردة. وهذا الوضع ينعكس في خوارزمية الإضافة (16) وخوارزمية تحديد الموقع (17) المصاحبة لها، واللتين نعرض تفاصيلهما فيما يلي:

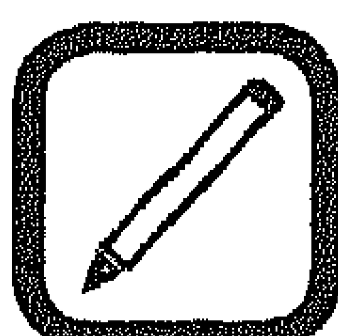
الخوارزمية (16):

```
crList* crList::insertCl(crList* list, int element)
    {crList*posn, *newNode=new crList;
     posn=list;
     posn=locatePos(list, posn, element);
     newNode->info=element;
     newNode->link = posn->link;
     posn->link=newNode;
     return list;
    }
```

الخوارزمية (17):

```
crList* crList::locatePos(crList* list, crList* posn,int element)
    {crList*previous,*next;
      bool found=false;
      previous = list;
      next = list->link;
      while (next !=list && !found)
              {if  (element < next->info)
          { //position is found
              posn = previous;
           found = true;
           } // endif
          else // update pointers
           {previous = next;
             next = next->link;
             } //end else
        } // endwhile
     posn = previous; //position is the last
     return posn;
     } // end  locatepos
```

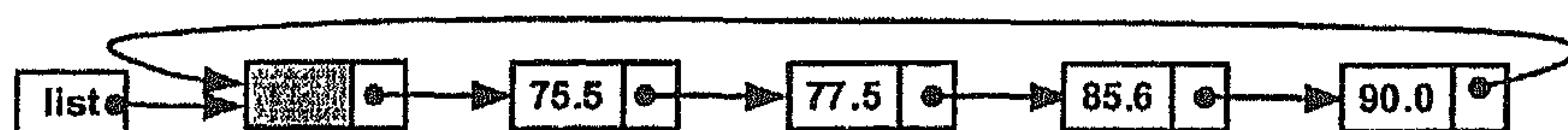
ولو قارنا بين هاتين الخوارزميتين وخوارزميتي الإضافة الأخريين الخاصتين بالقائمة المفردة لوجدنا أننا أسقطنا العديد من الجمل التي تضمنتها عملية إضافة عناصر جديدة إلى القائمة المفردة. ومن نتائج ذلك تبسيط العملية وزيادة درجة وضوح الخطوات التي تنطوي عليها عملية الإضافة. وكما ترى، فأياً كان موقع إضافة العنصر الجديد، فإن قيمة المؤشر (posn) العائدة من (locatePos) لن تكون (NULL) وأن عملية الإضافة لا بد وأن تنطوي، في كل مرة تتم فيها عملية الإضافة، على فك ارتباط العنصر السابق بما يشير إليه، حتى ولو كان لدينا عنصر حقيقي واحد فقط أو كان لدينا رأس القائمة فحسب.



تدريب (10)

افترض أن لدينا القائمة الدائرية التالية، ونريد أن نقوم بإضافة القيمة 82.5 في مكانها المناسب في هذه القائمة، فما عدد مرات تنفيذ كل من الجمل والإختبارات الشرطية التالية:

```
1.  if element < next-> info               // in locatePos
2.  next= next-> link;                     // in locatepos
3.  newNode-> link= posn-> link;           // in insertCl
```



الشكل (33): قائمة دائرية (تدريب (10))

4.1.5 حذف العناصر من القوائم الدائرية (Deletion)

كما هو الحال بالنسبة لعملية الإضافة، فإن عملية حذف أحد عناصر القائمة الدائرية ينطوي على إجراء تعديل على سلسلة المؤشرات، أياً كان موقع الحذف. وحتى لو كان العنصر المحذوف هو العنصر الأخير في القائمة فإن ذلك لن يؤدي إلى إزالة التركيب البياني بكامله. إذ يبقى لدينا في هذه الحالة رأس القائمة مشيراً إلى نفسه، على النحو الذي أوضحناه من قبل في الشكل (31). ويمكن أن نشير إلى القائمة في هذه الحالة بأنها خالية. وهذا على خلاف صورة القائمة الخالية في حالة عدم وجود الرأس، والتي يستدل عليها من خلال وجود القيمة NULL في المؤشر الخارجي. وفيما يلي نوضح كيف تتم عملية الحذف في القائمة الدائرية ذات الرأس، مع الافتراض بأن القيم مرتبة داخل هذه القائمة.

الخوارزمية (18):

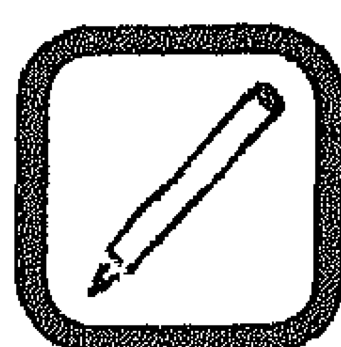
```
crList* crList::deleteCl(crList* list ,int element)
  { crList*pred, *pos;
   pos=list;
   pred=findPosn(list,pos,element ) ;
   pos=pred->link;
    if (pos==list)
    cout<<"element is not in list";
   else  //  update pointers and delete element
    {pred->link=pos->link;
     delete(pos);
    }// endelse
  return list;
  } // deleteCl
```

الخوارزمية (19):

```
crList* crList::findPosn(crList* list, crList* pos, int element )
 { crList  *current, *back, *pred;
   back=list;
   current=list->link;
   while (current!=list && current->info!=element)
    {back=current;
     current=current->link;
     } // end while
   pos=current; // position of deletion
   pred=back;  // previous location
  return pred;
  } // end findpos
```

ومرة أخرى نجد أنفسنا أمام وضع مبسط بالمقارنة مع خوارزمية الحذف الخاصة بالقوائم المفردة. فقد حذفنا العديد من الجمل، ولم يعد يهمنا أمر وجود العنصر المراد حذفه في أي موقع في القائمة (في البداية أو الوسط أو النهاية). والحالة الوحيدة التي بقيت على حالها تقريباً مع بعض التعديل على طريقة الاختبار، هي حالة عدم وجود العنصر في القائمة، كما يتضح من الجملة الشرطية في الدالة (deleteCl).

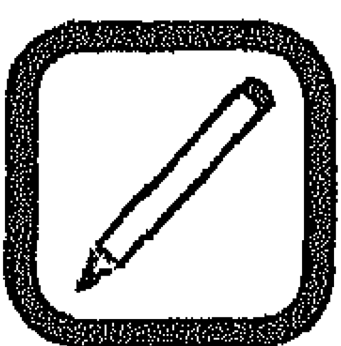
تدريب (11)



تأمل، مرة أخرى، الشكل المعطى في التدريب (10) المذكور سابقاً، واحسب عدد مرات تنفيذ الجمل التالية في حالة حذف العنصر المتمثل بالقيمة (90.0) من القائمة.

```
1.  back:= current;            // in findPos
2.  pos= current;              // in findPos
3.  pred->link=pos->link;      // in deleteCl
```

تدريب (12)



أعد كتابة خوارزميات الإضافة والحذف بحيث تنسجم مع الصيغة الثالثة للقوائم الدائرية حيث يكون المؤشر في وضع يشير فيه إلى العنصر الأخير في القائمة، وحيث ليس هناك رأس في القائمة وذلك على النحو الموضح في الجزء الثالث من الشكل (30).

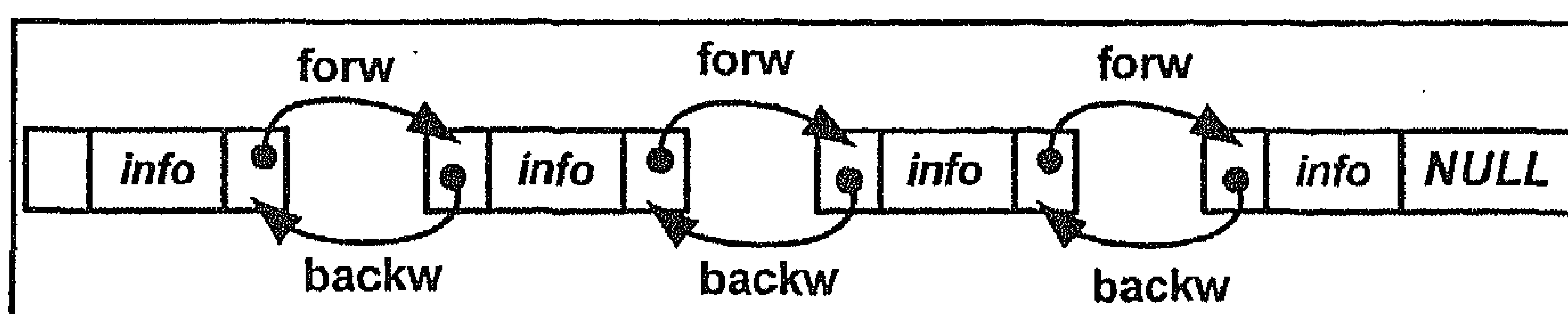
2.5 القوائم الثنائية وعملياتها (Doubly-linked Lists)

على الرغم من أن القوائم الدائرية تمتاز على القوائم المفردة الخطية ببعض السمات الهامة، فإنها هي الأخرى تعاني من بعض المشكلات أيضاً. فالمرء لا يستطيع أن يستعرض القائمة الدائرية بطريقة انعكاسية Backward ابتداء من العنصر الأخير أو أي نقطة معلومة في القائمة، كما أن المرء لا يستطيع أن ينجز عملية الحذف باستخدام مؤشر واحد فقط. ولو عدت إلى الخوارزمية (17) والخوارزمية المصاحبة لها، لوجدت أننا قد استعنا بمؤشرين اثنين لإجراء الحذف: الأول للإشارة إلى العنصر المطلوب حذفه والآخر للإشارة إلى العنصر السابق عليه. وفي الحالات التي تستلزم مثل هذه المتطلبات، فإن التركيب المناسب هو القائمة الثنائية (doubly-linked/two-way)

وتقوم فكرة القوائم الثنائية على وجود مؤشرين لكل عنصر في القائمة: الأول يشير إلى العنصر التالي، والآخر يشير إلى العنصر السابق. ومن هنا فإننا، على خلاف القوائم المفردة والقوائم الدائرية، نحتاج إلي وجود ثلاثة حقول على الأقل في تعريف القائمة الثنائية: أحدها لتخزين المعلومات، والثاني لتخزين المؤشر الأمامي (forward) والثالث لتخزين المؤشر الانعكاسي (backward)، كما يلي:

```
class dlList
{private:
    dlList* backw;
    int info;
    dlList* forw;
public:
dlList* creatDl(dlList*,dlList*, char element );
        ...
        ...
  };
...
...
void main()
{ dlList*list=new dlList;
...
...
}
```

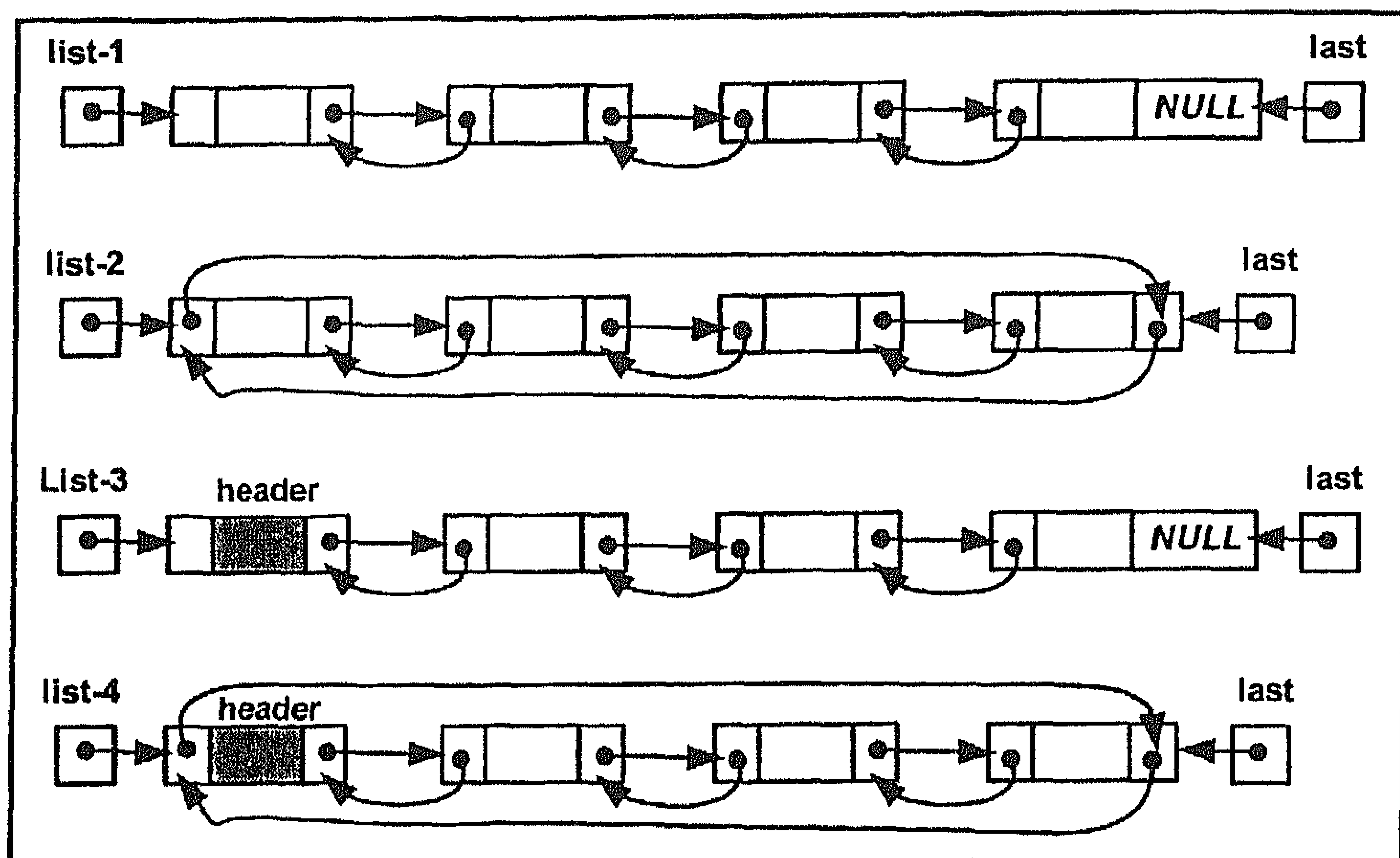
لاحظ أننا لم نحدد النمط البياني للحقل (info) وللمتغير (element) إذ يعتمد ذلك على نوع القيم التي نتعامل معها. وبناء على هذا المفهوم، فإن القائمة يمكن أن تتخذ الصورة الموضحة في الشكل (34).



الشكل (34): قائمة ثنائية.

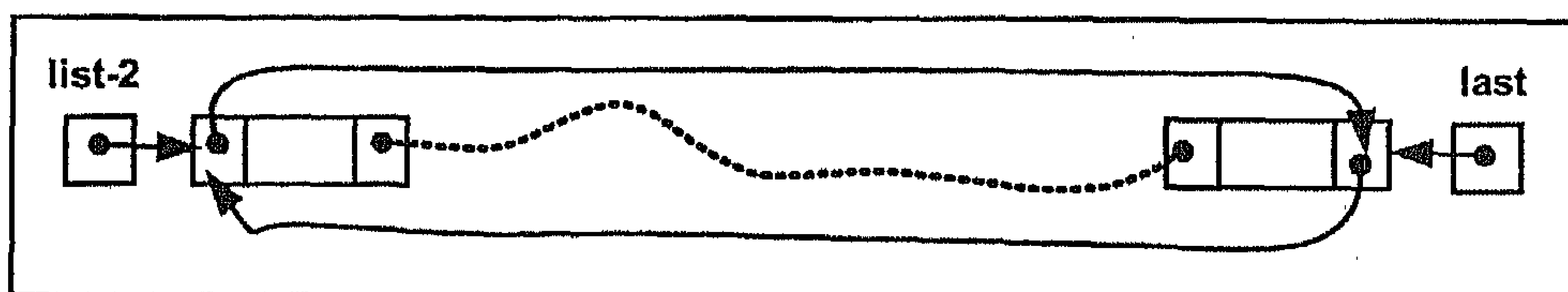
وبناء على هذا التصور فإن من الممكن، بالإضافة إلى ما ذكر أعلاه، أن نقوم باستعراض القائمة في الاتجاهين كليهما. ولكن لكي يتم ذلك لا بد من وجود العدد المناسب من المؤشرات. وبذلك نستطيع أن نتحرك في الاتجاهين من خلال تتبع سلسلة المؤشرات التي تربط بين العناصر.

وفي الحقيقة هناك أكثر من صيغة للقائمة الثنائية، اعتماداً على وجود رأس للقائمة أو على توفر الطبيعة الدائرية أم لا. والشكل (35) يوضح أهم الصيغ التي ترد عليها القائمة الثنائية.



الشكل (35): الصيغ المختلفة للقائمة الثنائية.

فالصيغة الأولى (list-1) تنسجم مع التعريف العام للقائمة الثنائية، وهي التي سنركز عليها في هذا الجزء من الوحدة. والصيغ الثلاثة الأخرى ما هي إلا تعديلات على هذه الصيغة العامة. والفرق بين الصيغة الأولى والصيغة الثانية هو في وجود مؤشر دائري على طرفي القائمة في الصيغة الثانية. وبذلك فإن قيمة NULL لم تعد تستعمل في هذه الصيغة. ومن هنا فإننا لو نظرنا إلى القائمة من بدايتها لوجدنا أنها دائرية، لأن مؤشر العنصر الأخير يشير إلى العنصر الأول. ولو حاولنا أن ننظر إلى القائمة من النهاية لوجدنا أنها تشكل قائمة دائرية أيضاً، لأن العنصر الأول يشير إلى الأخير في الوقت نفسه الذي يشير فيه إلى العنصر التالي. ويمكن توضيح هذه الخاصية الدائرية المزدوجة بالشكل (36).



الشكل (36): مقطع دائري لقائمة ثنائية.

أما الصيغة الثالثة فيميزها عن الصيغة الأولى وجود رأس للقائمة. وفيما عدا ذلك فإن الأمور كلها سواء. وأخيراً فإن الصيغة الرابعة تختلف عن الأولى في أنها دائرية وتضم عنصراً خاصاً في بدايتها كما هو الحال في الصيغة الثالثة. والصيغة الأخيرة للقائمة الثنائية هي أكثر الصيغ تعقيداً نظراً للإمكانات التي توفرها.

وعلى الرغم من الأهمية الكبيرة لهذه الصيغ المختلفة التي ترد بها القوائم المتصلة الثنائية فإن المجال لا يتسع هنا للحديث عن العمليات المتعلقة بها جميعاً. وسيقتصر حديثنا على الصيغة العامة (list-1).

1.2.5 بناء القوائم الثنائية (doubly-linked list creation)

إن بناء القوائم المتصلة، على النحو الذي بيناه فيما سبق، ينطبق على حالة القوائم الثنائية، مع وجود إضافتين أساسيتين على ما تم ذكره من قبل:

الأولى: إجراء التعديل المناسب على سلسلة المؤشرات الأمامية والخلفية معاً.

الثانية: إجراء التعديل المناسب على المؤشر الخارجي الذي يشير إلى نهاية القائمة.

فالقائمة تبدأ خالية، ويستدل على خلوها من خلال قيمة المؤشرات الخارجية، ثم نبدأ بإضافة العناصر إليها واحداً تلو الآخر، بالأسلوب نفسه الذي وضحناه سابقاً. والخوارزمية (20) تقدم تفاصيل القيام بهذه العملية.

الخوارزمية (20):

```
void dlList::creatDl(dlList* list,dlList* last, int element)
 { dlList*nextNode= new dlList;

   nextNode->info=element;
   nextNode->forw=NULL;
   nextNode->backw=NULL;
   *list=*nextNode;
   *last=*nextNode;

 }
```

وكما تلاحظ، فإن عملية البناء هي بحد ذاتها عملية إضافة، ولكنها في هذه الحالة إضافة إلى آخر القائمة فقط. ويمكن أن يتم بناء القائمة أيضاً عن طريق الإضافة إلى بدايتها فقط، وهذا يعتمد على نوع التطبيق المعني. وستتضح هذه العلاقة بشكل أكبر فيما بعد، عند الحديث عن خوارزمية الإضافة (22).

2.2.5 استعراض القوائم الثنائية (doubly-linked list traversal)

إن إحدى المميزات الأساسية للقائمة الثنائية، كما ذكرنا من قبل، هي توفير إمكانية القيام باستعراض عناصرها في الاتجاهين كليهما. فمن الممكن أن نبدأ من بداية القائمة ونسير باتجاه نهاية القائمة حتى ننتهي من جميع العناصر. ومن الممكن أيضاً أن نبدأ من آخر عنصر في القائمة، مستفيدين بذلك من وجود المؤشر الخارجي last، ونسير باتجاه عكسي نحو بداية القائمة متتبعين سلسلة المؤشرات الإنعكاسية (backw) حتى نصل إلى العنصر الأخير في هذا الاتجاه. ويمكن أن نعبر عن هذه العملية بالخوارزمية التالية.

الخوارزمية (21):

```
void dlList::traverseDl(dlList* list,dlList* last,char direction)
  {// traversing a doubly-linked list in either direction
   dlList *current;
            if (direction=='f') // forw traversal
   {current=list;
    while (current !=NULL)
     {process(current->info);
            current=current->forw;
     } // end while
   } // end if
  else  // backw traversal
   {current=last;
    while (current !=NULL)
            { cout<<current->info<<" ";
      current=current->backw;
     }
   }// end else
  }//  traverseld
```

ولعلك تلاحظ، عزيزي الدارس، بأننا قد ضمنّا هذه الخوارزمية متغيراً جديداً هو (direction). وهو يشتمل على حرف واحد أما "f" أو "b" ومهمته تحديد اتجاه استعراض العناصر. وفي ضوء ذلك فقد اشتملت الخوارزمية على جملة شرطية للتحقق من قيمة هذا المتغير، وبذلك تم تخصيص الجزء الأول للاستعراض المعتاد باتجاه نهاية القائمة (forw) وتم تخصيص البديل (else) للاستعراض الانعكاسي باتجاه بداية القائمة (backw).

3.2.5 إضافة عناصر جديدة إلى القوائم الثنائية (insertion)

إن الاختلاف بين القوائم الثنائية والأنواع الأخرى من القوائم المتصلة يكمن في ضرورة التعامل مع الإشارة المزدوجة للعنصر الواحد إلى الأمام وإلى الخلف. فإذا كان لدينا العنصر المتمثل بالقيمة x ونرغب في إضافته إلى القائمة الثنائية (list) في الموضع posn، فإننا يمكن أن نقوم بهذه العملية وفقاً لأحد الأوضاع التالية:

أولاً: الإضافة بين عنصرين آخرين

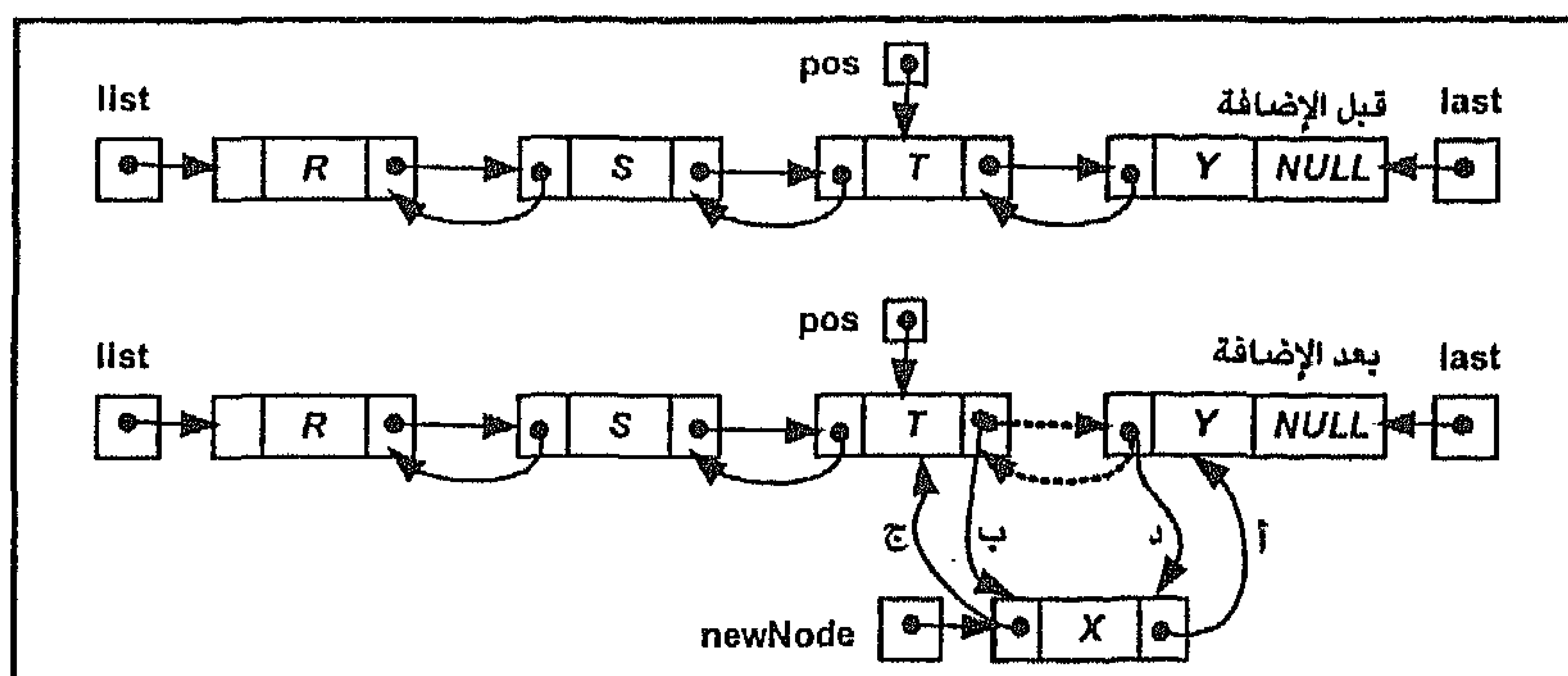
تتم عملية الإضافة بين عنصرين آخرين على النحو الموضح في الشكل (37) وذلك وفقاً للخطوات التالية، مع ملاحظة أننا قد قمنا بوضع أرقامها على الشكل نفسه وذلك زيادة في التوضيح (يمثل العنصر المراد إضافته بالقيمة X والعنصر السابق بالقيمة T والعنصر التالي بالقيمة Y).

أ. ربط العنصر الجديد بالعنصر التالي له والمتمثل بالقيمة "Y" وذلك من خلال إسناد قيمة مؤشر العنصر السابق المشار إليه بالمؤشر (posn) إليه، هكذا:

newNode->forw=posn->forw;

ب. فك ارتباط المؤشر (forw) الخاص بالعنصر السابق المتمثل بالقيمة "T" بالعنصر التالي المتمثل بالقيمة "Y"وذلك من خلال تحويل اتجاهه إلى العنصر الجديد، هكذا:

posn->forw=newNode;



الشكل (37): إضافة عنصر جديد إلى قائمة ثنائية.

ج. ربط العنصر الجديد بالسابق المتمثل بالقيمة "T" وذلك من خلال إسناد قيمة المؤشر الداخلي للعنصر التالي، هكذا:

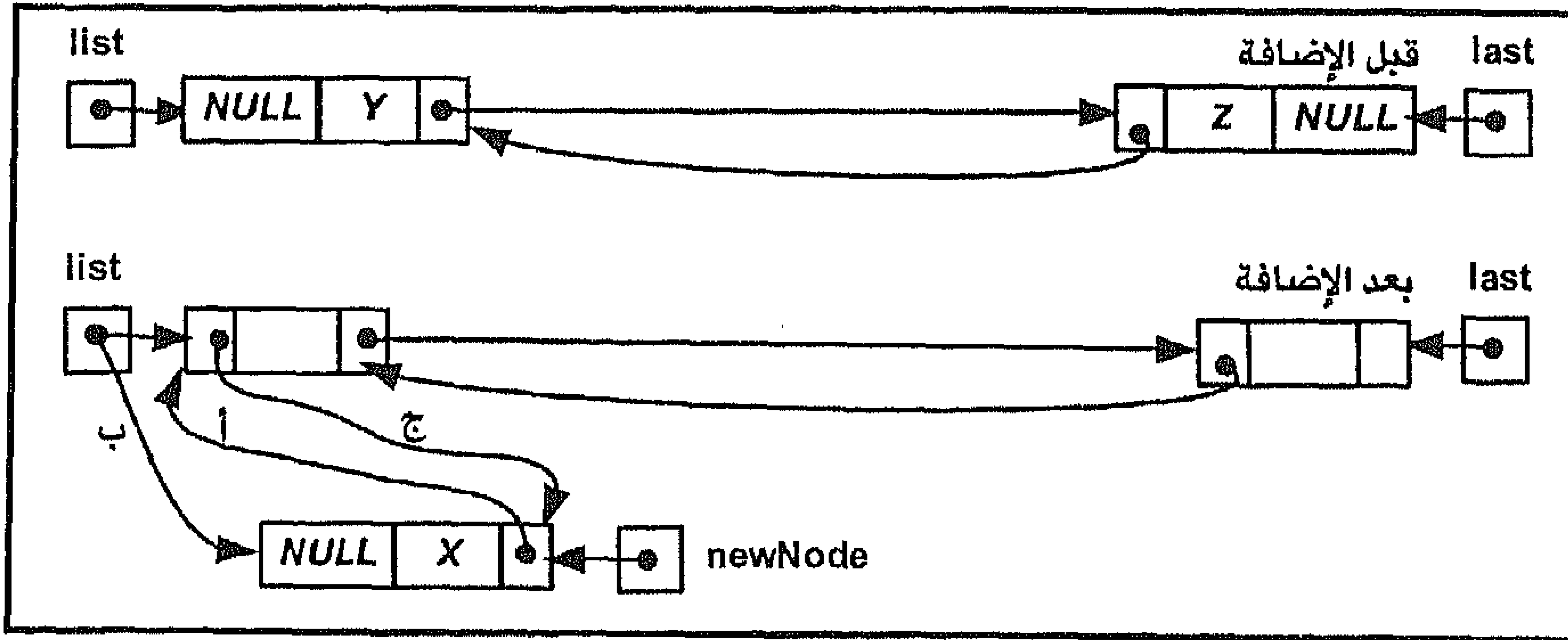
newNode->backw->forw=newNode;

د. فك ارتباط المؤشر الخلفي (BACKW) للعنصر التالي بالعنصر السابق المتمثل بالقيمة "T" وذلك من خلال تحويل اتجاهه إلى العنصر الجديد، هكذا:

newNode->forw->backw=newNode;

ثانياً: الإضافة إلى بداية القائمة الثنائية

لعلك لاحظت بأن العنصر الجديد قد جاء بين العنصرين الممثلين بالقيمتين «T" و"Y" لأنه يحتل هذا الموقع بحكم ترتيبه الهجائي ضمن التسلسل المعتمد في القائمة. والآن ماذا لو كان موقع العنصر المرغوب إضافته هو بداية القائمة؟ هل سيؤدي ذلك إلى اختلاف في أسلوب العمل؟ كما ترى، فإن قيمة أحد المؤشرين في العنصر الأول هي NULL. وهذا العنصر، بحكم موقعه، مرتبط بالمؤشر الخارجي list. ومعنى ذلك أننا ينبغي أن نتعامل مع بعض الظروف المختلفة عن مسألة الإضافة بين عنصرين، على النحو الذي بيناه أعلاه. وفي ضوء هذه الظروف الجديدة، فإن عملية الإضافة إلى بداية القائمة يمكن أن تتم على النحو الموضح في الشكل (38) وفقاً للخطوات التالية (مع ملاحظة أننا فعلنا هنا ما فعلناه في الشكل (37) السابق من حيث وضع أرقام الخطوات على الشكل من أجل زيادة التوضيح):



الشكل (38): الإضافة إلى بداية القائمة الثنائية.

أ. ربط المؤشر الأمامي (forw) للعنصر الجديد بالعنصر التالي له المتمثل بالقيمة "Y" وذلك من خلال جملة الإسناد التالية:

newNode->forw=list;

ب. فك ارتباط المؤشر الخارجي (list) بالعنصر الذي يشير إليه وتحويل اتجاهه إلى العنصر الجديد، كما يلي:

list=newNode;

ج. تعديل قيمة المؤشر الخلفي (backw) للعنصر المتمثل بالقيمة "Y" من NULL إلى حيث يصبح في وضع يشير فيه إلى العنصر الجديد، كما يلي:

newNode->forw->backw=newNode;

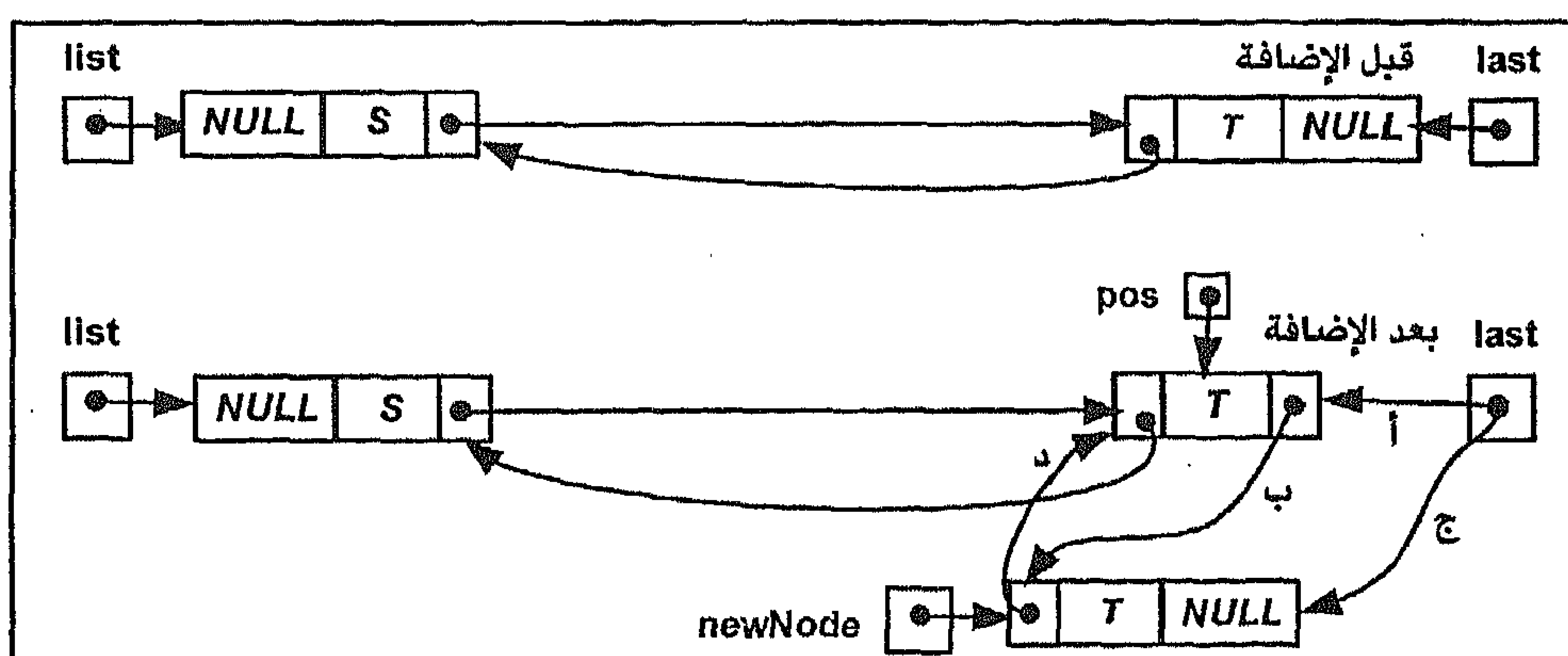
د. تعديل قيمة المؤشر الخلفي (backw) للعنصر الجديد بحيث تصبح NULL، كما يلي:

newNode->backw=NULL;

ولعلك، عزيزي الدارس، تلاحظ بأننا لم نشر في جميع هذه الخطوات إلى المؤشر الخارجي (posn) مطلقاً. والسبب وراء ذلك هو أن مهمة هذا المؤشر هي الإشارة إلى العنصر السابق لموضع العنصر الجديد. وطالما أنه لا يوجد مثل هذا العنصر، فإن قيمته NULL وهذه القيمة هي الدليل على الإضافة إلى البداية.

ثالثاً: الإضافة إلى نهاية القائمة الثنائية

إن الإضافة إلى نهاية القائمة الثنائية لا تكاد تختلف كثيراً عن الحالة السابقة. فالآن نحن بصدد التعامل مع المؤشر الخارجي (last) ومع المؤشر الداخلي الأمامي (forw) للعنصر الأخير في القائمة. ويمكن أن تتم عملية الإضافة في هذه الحالة علي النحو الموضح في الشكل (39)، وتتسلسل الخطوات كما يلي (مع ملاحظة الترقيم المقابل لهذه الخطوات على الشكل نفسه).



الشكل (39): إضافة عنصر جديد إلى نهاية القائمة الثنائية.

أ. تعديل قيمة المؤشر الأمامي (forw) للعنصر الجديد بحيث تصبح NULL، كما يلي:

newNode->forw=NULL;

ب. تعديل قيمة المؤشر الأمامي (forw) للعنصر الأخير في القائمة بحيث يصبح في وضع يشير فيه إلى العنصر الجديد، كما يلي:

posn->forw=newNode;

ج. فك ارتباط المؤشر الخارجي (last) بالعنصر الأخير وتحويل اتجاهه إلى العنصر الجديد كما يلي:

Last=newNode;

د. تعديل قيمة المؤشر الخلفي (backw) للعنصر الجديد بحيث تصبح في وضع الإشارة إلى العنصر السابق المتمثل بالقيمة "T"، كما يلي:

newNode->backw=posn;

أما وقد رأينا كيف نقوم بعملية الإضافة في الحالات الثلاثة السابقة، فقد حان الوقت الآن لكي نجمل هذه الحالات المختلفة في خوارزمية واحدة. وتذكر، على أية حال، بأن العنصر المضاف قد يكون العنصر الوحيد في القائمة، وبالتالي فإن كلا المؤشرين الخارجيين (list) و(last) سيشيران إلى هذا العنصر. والخوارزمية (insertDl) تعرض تفاصيل القيام بعملية الإضافة.

الخوارزمية (22):

```
void dlList::insertDl(dlList* listt,dlList* lastt,char element)
  {//inserting an element into a sorted doubly-linked list
   dlList*posn,*list,*last, *newNode=new dlList;
   List=listt;
   last=lastt;
   newNode->info=element;
   posn=locatePos(list, element);
    if (posn==NULL) //insert in the beginning
    {newNode->forw=list;
    list=newNode;
     if (last==NULL) // list is empty
     last=newNode;
     else  //insert before the first element
    newNode->backw=NULL;
    } // end if
   else // insert somewhere other than the beginning
         {newNode->forw=posn->forw;
     newNode->backw=posn;
     posn->forw=newNode;
     if (last==posn||newNode->info>last->info)// insert at end
       last=newNode;
      else //insert between two elements
```

```
{newNode->forw->backw=newNode;
 newNode->backw=posn;
 }
}//  end els
*listt=*list;
*lastt=*last;
}// end insertdl
```

الخوارزمية (23):

```
dlList* dlList::locatePos(dlList* temp,char element)
{//finding a position for insertion
 dlList *previous,*next,*pos;
 bool found;
 if (temp==NULL ||element<temp->info)
        pos= NULL;
 else
  {previous=temp;
       next =temp->forw;
   found=false;
   while (next!=NULL && !found)
       {if (element<next->info)
      {pos=previous;
       found=true;
      }
      else
       {previous=next;
        next=next->forw;
        }
      }
      pos = previous;
    }
   return pos;
   }
```

ولعلك لاحظت أن خوارزمية الإضافة قد تضمنت استدعاء الخوارزمية (locatePos)، التي تشبه إلى حد كبير الخوارزمية (3) التي سبق تعريفها. لاحظ أيضاً بأن التركيب الشرطي (if-else) في الخوارزمية (22) يأخذ بالاعتبار جميع الأوضاع التي يمكن أن تتم بها عملية الإضافة، كما تشتمل على جمع الخطوات الأربعة التي عرضناها لدى الحديث عن الأوضاع الثلاثة المختلفة لعملية الإضافة.

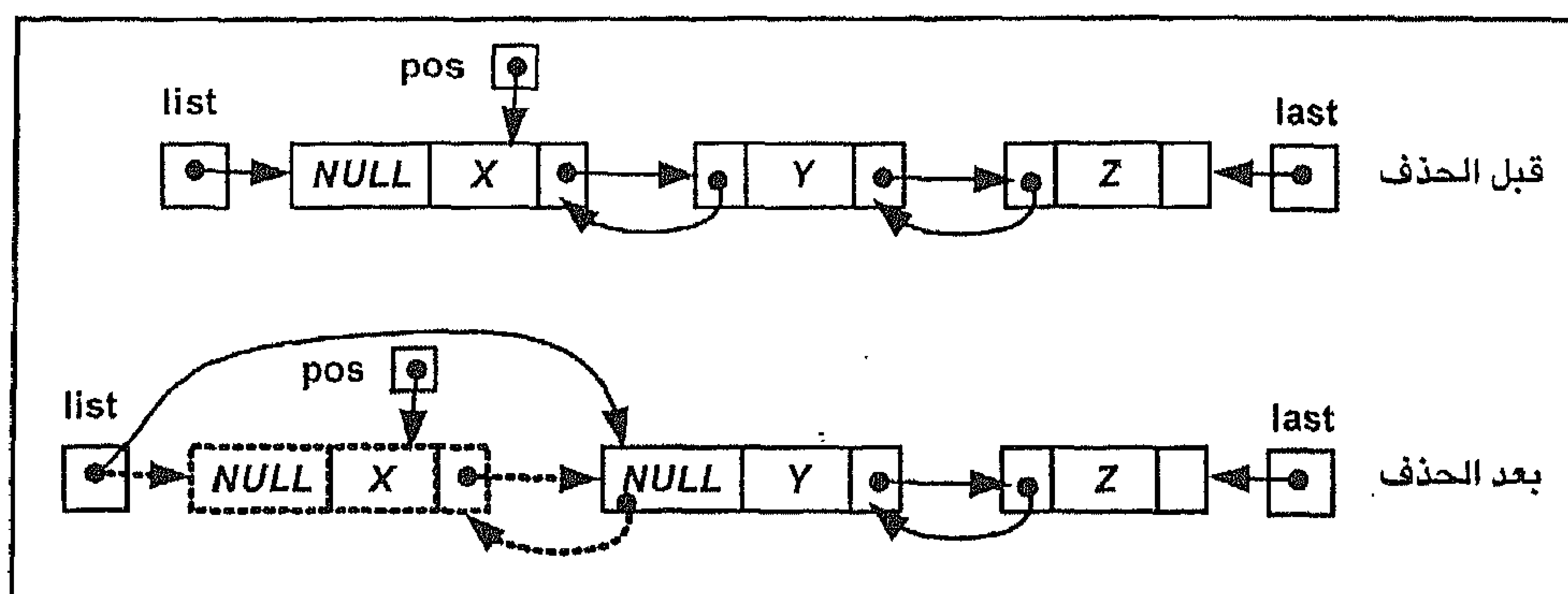
4.2.5 حذف العناصر من القوائم الثنائية (deletion)

إن عملية الحذف، شأنها شأن عملية الإضافة، تتأثر بالوضع الخاص بالعنصر المراد حذفه. فقد يكون هذا العنصر في بداية القائمة، وقد يكون في وسطها، وقد يكون في

نهايتها، وقد لا يكون فيها مطلقاً. ولعلك تذكر أننا قد ناقشنا هذه الأوضاع الأربعة عند حديثنا عن القوائم الفردية. ولا يكاد يختلف الأمر هنا عن سابقه، إلا في مسألة عدد المؤشرات التي تستدعي التعديل. وفيما يلي نستعرض الخطوات التي نسير فيها لحذف أحد العناصر في الأوضاع المختلفة، مع ملاحظة أن أرقام هذه الخطوات قد تم وضعها وفقاً لتسلسلها على المؤشرات في الأشكال التوضيحية المصاحبة.

أولاً: حذف عنصر من البداية

افترض أننا نرغب في حذف العنصر "X" من القائمة الثنائية (list) وافترض أن هذا العنصر يقع في بداية القائمة، على النحو الموضح في الشكل (40)، فإن عملية الحذف ستتخذ الخطوات التالية:



الشكل (40): حذف عنصر من بداية القائمة الثنائية.

أ. فك ارتباط المؤشر الخارجي (list) بالعنصر المراد حذفه المتمثل بالقيمة "X" وتحويل اتجاهه إلى العنصر التالي المتمثل بالقيمة "Y" كما يلي:

list= posn->forw;

ب. فك ارتباط المؤشر الخلفي للعنصر التالي المتمثل بالقيمة "Y" وتغيير قيمته إلى NULL لأنه سيصبح العنصر الأول في القائمة بعد إتمام عملية الحذف، كما يلي:

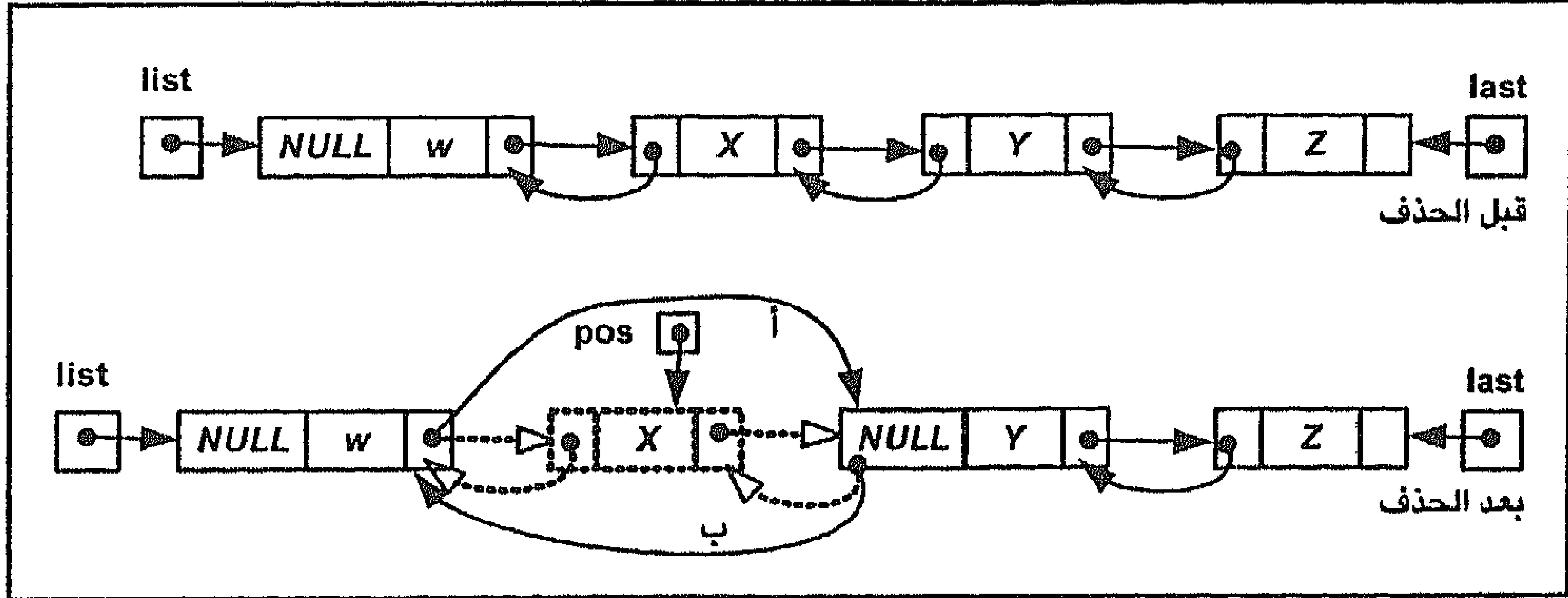
list->backw= NULL;

ج. إزالة العنصر المرغوب حذفه كلية من الذاكرة، واستعادة المكان المحجوز إلى قائمة الأماكن الخالية في الذاكرة، وذلك من خلال الجملة التالية:

delete(posn);

ثانياً: حذف عنصر من وسط القائمة

دعنا نرى الآن، كيف يمكن أن نقوم بعملية الحذف إذا كان العنصر المرغوب حذفه واقعاً بين عنصرين آخرين، على النحو الموضح في الشكل (41).



الشكل (41): حذف عنصر من القائمة الثنائية واقع بين عنصرين آخرين.

أ. فك ارتباط المؤشر الأمامي (forw) للعنصر السابق المتمثل بالقيمة "W" بالعنصر المراد حذفه وتحويل اتجاهه إلى العنصر التالي المتمثل بالقيمة "Y" كما يلي:

posn->backw->forw=posn->forw;

ب. فك ارتباط المؤشر الخلفي (backw) للعنصر التالي المتمثل بالقيمة "Y" بالعنصر المراد حذفه وتحويل اتجاهه إلى العنصر السابق المتمثل بالقيمة "W" كما يلي:

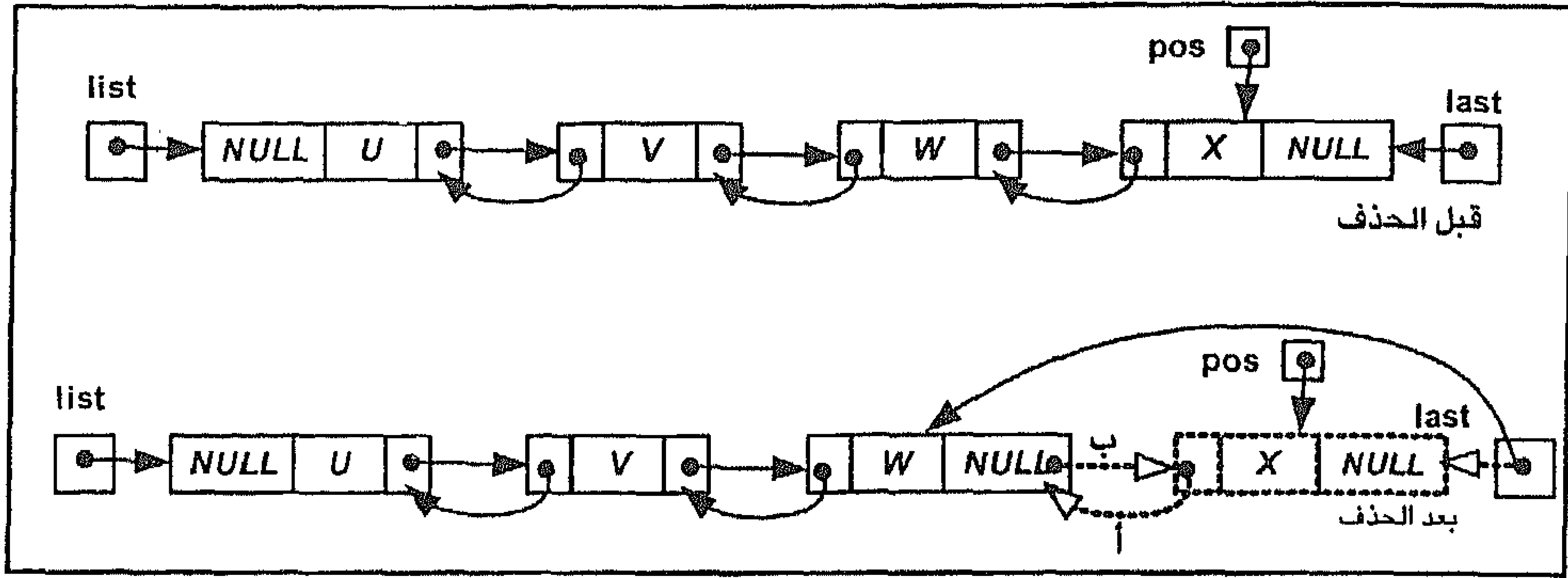
posn->forw->backw = posn->backw;

ج. إزالة العنصر المرغوب حذفه كلية من الذاكرة واستعادة المكان المحجوز إلى قائمة الأماكن الخالية وذلك من خلال الجملة التالية:

delete(posn);

ثالثاً: حذف عنصر من نهاية القائمة الثنائية

كما هو الحال بالنسبة لحذف العنصر الأول في القائمة، فإننا بصدد التعامل مع مؤشر خارجي هو (last). وبذلك يمكن التعبير عن خطوات الحذف على النحو التالي، كما هو موضح في الشكل (42):



الشكل (42): حذف آخر عنصر في القائمة.

أ. فك ارتباط المؤشر الخارجي (last) بالعنصر المراد حذفه المتمثل بالقيمة "X" وتحويل اتجاهه إلى العنصر السابق المتمثل بالقيمة "W" كما يلي:

last=posn->backw;

ب. فك ارتباط المؤشر الأمامي للعنصر السابق المتمثل بالقيمة "W" وتحويل القيمة إلى NULL لأنه سيكون العنصر الأخير بعد إتمام عملية الحذف، كما يلي:

last->forw= NULL;

ج. إزالة العنصر المرغوب حذفه كلية من الذاكرة، واستعادة المكان المحجوز إلى قائمة الأماكن الخالية وذلك من خلال الجملة التالية:

delete(posn);

والآن، بعد أن أوضحنا كيف نقوم بعملية الحذف في الحالات المختلفة، دعنا نجمل هذه الأوضاع والخطوات التي انطوت عليها في خوارزمية واحدة للحذف تأخذ بعين الاعتبار جميع أوضاع الحذف من القائمة الثنائية.

الخوارزمية (24):

```
void dlList::deleteDl(dlList* listt ,dlList* lastt,char element)
  { //        deleting an element from a doubly-linked list
   dlList *posn, *list=listt,*last=lastt;
   posn=findPosn (list, element); //find position of element
        if (posn==NULL)
    cout<<"element is not found in list";
   else
       if (posn==list)  // first element in list
      {
      list= posn->forw;
      list->backw= NULL;
```

```
  if (list==NULL)
    last=NULL; // one element
   }
  else
   if (posn->info==last->info)  //  last element
    {last=posn->backw;
     last->forw= NULL;
     }
    else // between two other elements
     {posn->backw->forw= posn->forw;
      posn->forw->backw = posn->backw;
      }
  delete(posn);
*listt=*list;
*lastt=*last;

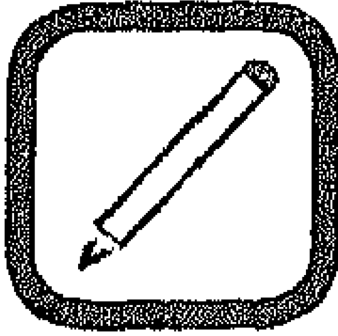
}// end deletedl
```

الخوارزمية (25):

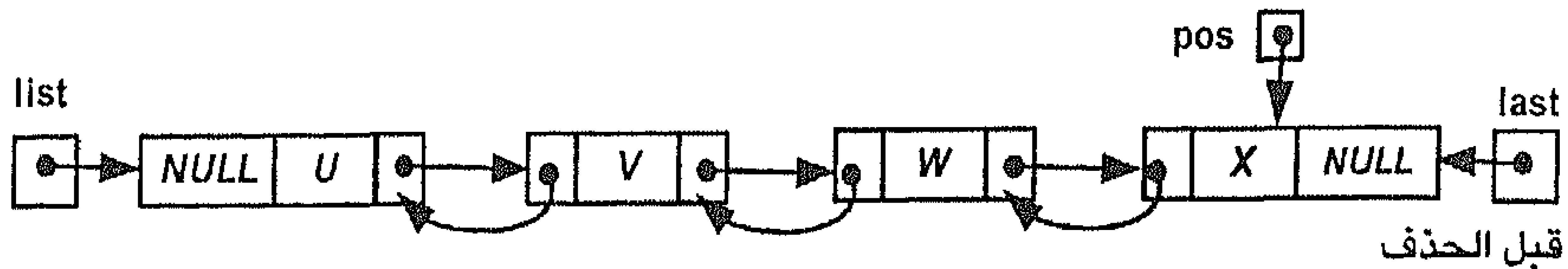
```
dlList* dlList::findPosn(dlList* list,char element )
 { //finding the position of an element in a doubly-linked list
  dlList  *current,*posn;
     if (list==NULL)// list empty
     posn = NULL;
   else
    if (list->info==element) // first item
     posn=list;
    else  // search for position
      {current = list;
       while (current !=NULL && current->info !=element)
         current =current->forw;
       posn = current;
      }//  end else
   return posn;
  }// end      findpos
```

ولو حاولنا أن نقارن بين الخوارزمية (25) والخوارزمية المماثلة لها رقم (5) لوجدنا أن الفرق بينهما طفيف جداً. وهذا الفرق يتمثل بصفة خاصة في استعمال مؤشر واحد لتحديد موقع العنصر المراد حذفه بدلاً من مؤشرين (posn, pred) في الحالة السابقة. ومرد ذلك هو الطبيعة الإشارية المزدوجة لكل عنصر من عناصر القائمة الثنائية، حيث لم نعد بحاجة إلى مؤشر آخر مساعد للإشارة إلى العنصر السابق.

تدريب (13)



تأمّل القائمة الثنائية المبينة في الشكل (43) ثم أجب على الأسئلة الواردة أدناه من خلال استخدام الخوارزمية (24)

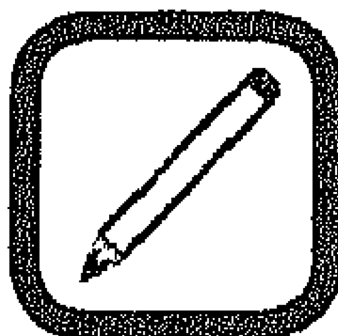


الشكل (43)

أ. افترض أن العنصر المراد حذفه من هذه القائمة هو العنصر الذي يحمل القيمة «E"، فأي من جمل الإسناد الواردة في الخوارزمية (24) سيتم تنفيذها في هذه الحالة؟

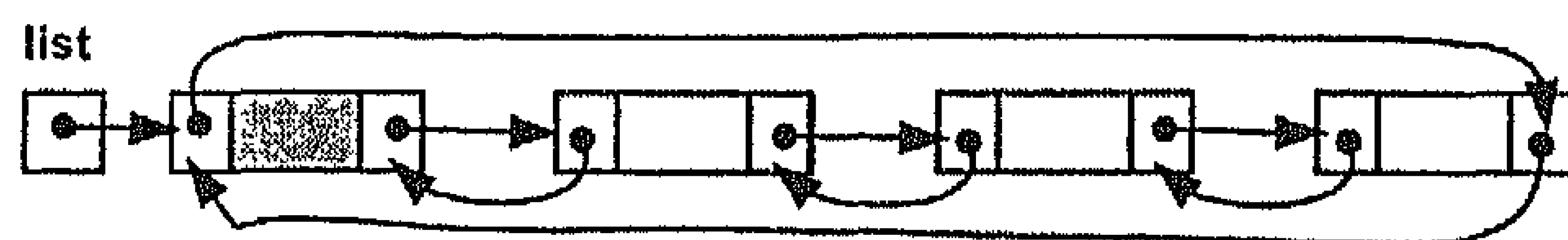
ب. افترض أننا نرغب في حذف العنصر الذي يحمل القيمة «R»، فأيّ من جمل الإسناد سيتم تنفيذها في الخوارزمية (24) في هذه الحالة؟

ج. افترض أننا نرغب في حذف العنصر الذي يحمل القيمة "N"، فأي من جمل الإسناد الواردة في الخوارزمية (24) سيتم تنفيذها في هذه الحالة؟



تدريب (14)

من الممكن تبسيط خوارزميتي الإضافة والحذف السابقتين إلى درجة كبيرة من خلال تضمين رأس للقائمة، وجعلها قائمة دائرية على النحو المبين في الشكل (44). والمطلوب هو أن تعيد كتابة الخوارزميتين بالاستعانة بالرسم المعطى، مع ملاحظة أننا في حالة الإضافة نستخدم مؤشرين هما: (previous) للإشـــارة إلى العنصر السابق على موضع الإضافة و(next) للإشارة إلى العنصر التالي لموضع الإضافة، وفي حالة الحذف نستخدم مؤشراً واحداً فقط هو (posn) للإشارة إلى العنصر المراد حذفه.



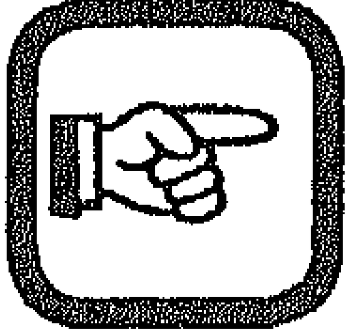
الشكل (44): قائمة ثنائية دائرية وذات رأس.

3.5 القوائم المشتركة وعملياتها (Multi-linked lists)

اعتمدت جميع صيغ القوائم المتصلة التي مرّ ذكرها حتى الآن على وجود مجموعة من العناصر المرتبطة معاً إما بمؤشر واحد، كما هو الحال في القوائم المفردة والقوائم الدائرية، أو بمؤشرين اثنين، كما هو الحال في القوائم الثنائية. وبينما لا يسمح النوع الأول من الربط بأكثر من اتجاه واحد للقائمة، وبالتالي فإن العنصر الواحد يحتل موقعاً واحداً في السياق المعتمد، فإن الأسلوب الآخر في الربط يسمح بوجود اتجاهين متقابلين وبالتالي فإن العنصر الواحد يحتل موقعاً مزدوجاً في السياقين الحادثين.

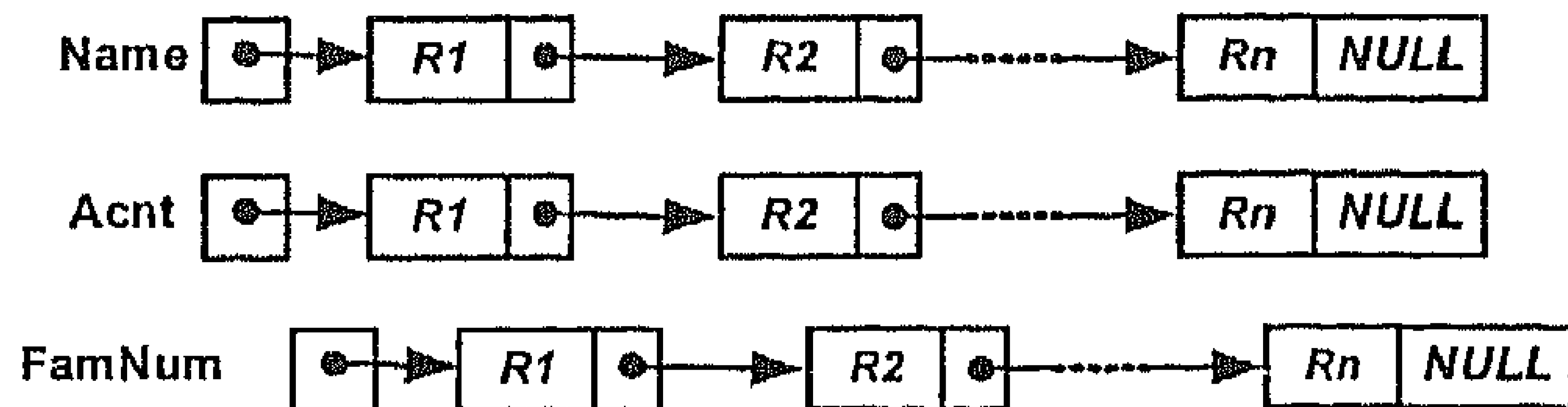
وعلى الرغم من أن القوائم الثنائية توفر إمكانية إضافية على الأنواع الأخرى من القوائم المتصلة، إلا أن ازدواجية الاتجاه فيها لا يغير كثيراً من منطق تكوين التركيب البياني. وكل ما في الأمر هو أن بإمكاننا أن نعكس الاتجاه، بحيث نستطيع أن ندخل على القائمة من آخرها أيضاً. فمن الممكن، مثلاً، استعراض العناصر باتجاه عكسي، ومن الممكن أيضاً أن نبحث عن أحد العناصر في التركيب البياني ابتداء من نهايته، إذا علمنا مسبقاً أن العنصر أقرب للنهاية منه للبداية. وعلى هذا الأساس، فإن التغير في موقع العنصر هو تغير محدود. فقد يكون في هذا الاتجاه وقد يكون في الاتجاه المقابل، وليس أكثر من ذلك.

ولكن افترض أن العنصر الواحد يظهر في قائمتين مختلفتين أو أكثر، فهل تصلح أي من الصيغ السابقة لتمثيل هذه العلاقة؟ وقبل أن نجيب على هذا التساؤل، دعنا نتأمل المثال التالي.



مثال (10)

افترض أن أحد البنوك يرغب في الاحتفاظ لعملائة بسجلات منظمة على نحو يمكنه بسهولة من الوصول إلى سجل كل عميل حسب اسمه، وحسب رقم حسابه، وحسب رقم دفتر العائلة. فإن إحدى الطرق لتنظيم هذه السجلات هي أن يكون لدينا ثلاث قوائم منفصلة: واحدة مرتبة هجائياً حسب أسماء العملاء، وأخرى مرتبة حسب أرقام الحسابات، وثالثة مرتبة حسب أرقام الأحوال المدنية، وذلك على النحو المبين في الشكل (45).



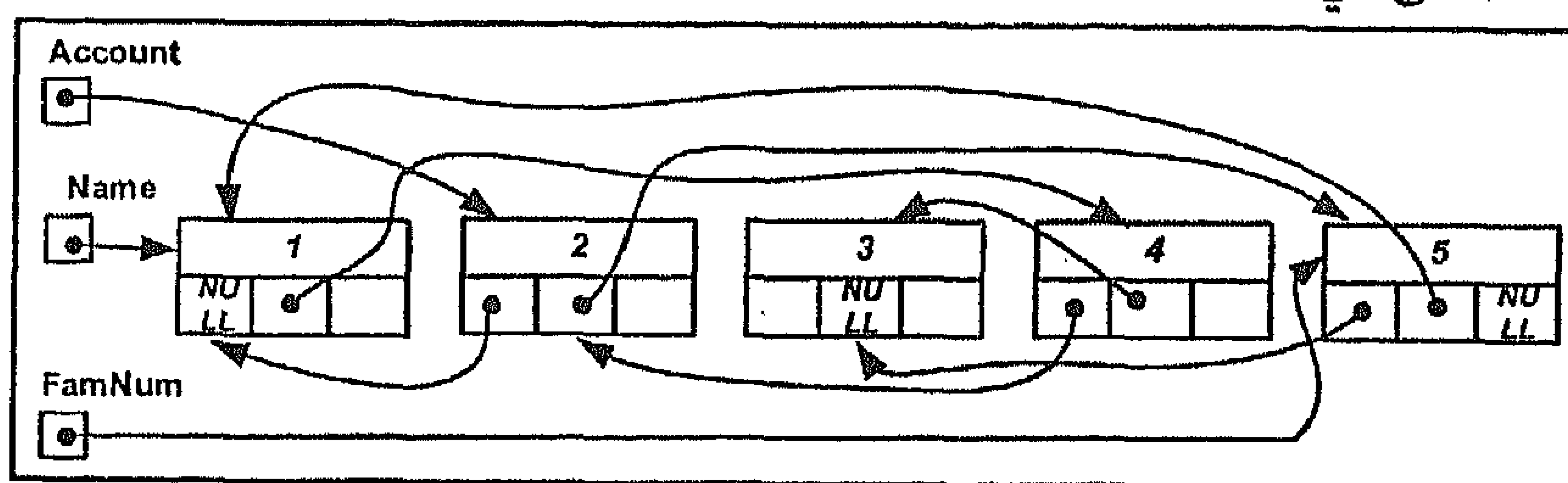
الشكل (45): مجموعة من السجلات مرتبة وفق سياقات مختلفة في قوائم منفصلة

فكما تلاحظ، ينطوي هذا الأسلوب على هدر في المساحة المتوفرة للتخزين. وبدلاً من الاحتفاظ بثلاث قوائم منفصلة، يمكننا أن نحتفظ بقائمة واحدة فقط، ونجعل كل مؤشر يشير إلى ما يليه في هذه السياقات الثلاث المذكورة أعلاه من خلال وجود العدد المناسب من المؤشرات. ولكي نوضح كيف يتم ذلك، دعنا نتأمل الجدول (1) التالي ولنر كيف يمكن تمثيله في قائمة واحدة.

جدول (1): أسماء عملاء أحد البنوك (المثال 10)

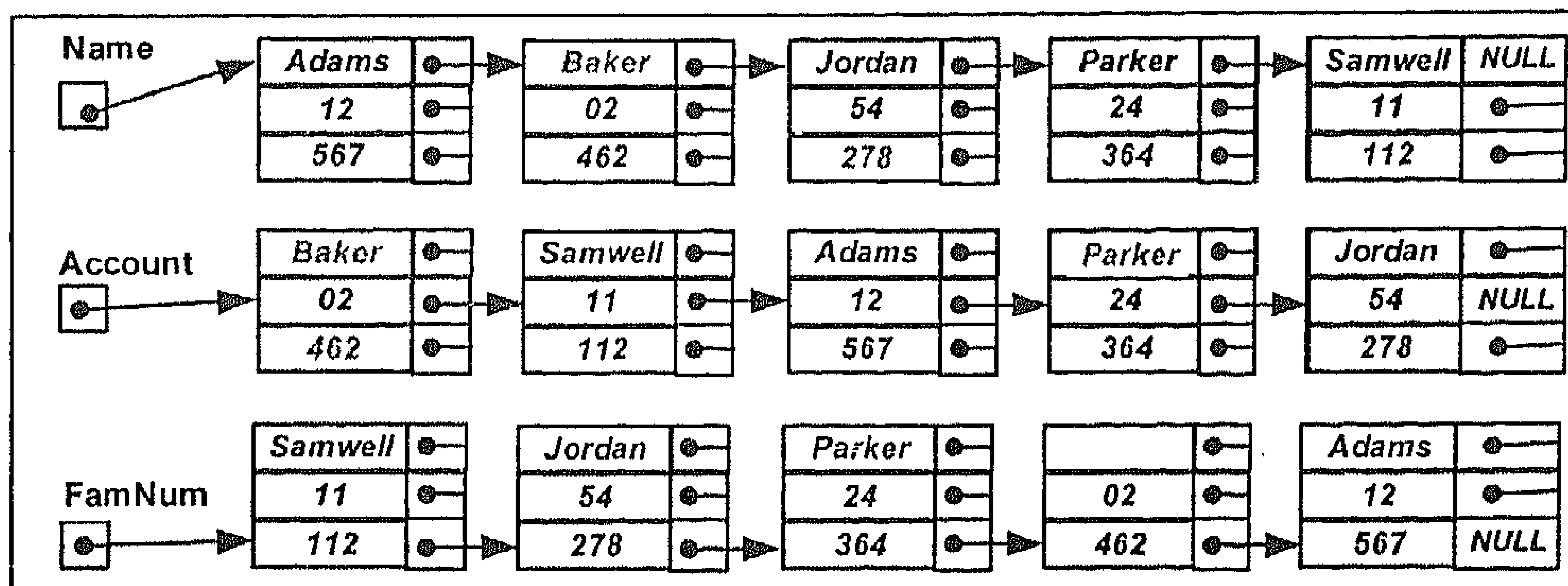
	CLIENT NAME	ACCOUNT NUMBER	FAMILY NUMBER	ACCOUNT INFO
1	Adams	12	567	...
2	Baker	02	462	...
3	Jordan	54	278	...
4	Parker	24	364	...
5	Samwell	11	112	...

فكما تلاحظ، هذا الجدول مرتب هجائياً وفق أسماء المودعين. ولكي نحتفظ بتسلسل سجلات هؤلاء المودعين وفق أرقام حساباتهم وأرقامهم العائلية، بالإضافة إلى التسلسل القائم وفقاً للأسماء، فإن من الممكن تمثيل هذا الجدول على هيئة قائمة متصلة، وهي على النحو الموضح في الشكل (46).



الشكل (46): قائمة ذات سياقات ترتيبية متعددة

وبناء على هذه القائمة الموضحة في الشكل (46)، فإن لدينا ثلاثة سياقات ترتيبية متميزة، يشترك فيها العنصر الواحد جميعا. ويمكن أن نبين العلاقات الموجودة بين العناصر في هذه السياقات المختلفة على النحو الموضح في الشكل (47).



الشكل (47): السياقات الثلاثة الممثلة في القائمة المتصلة الموضحة في الشكل (46).

ولعلك لاحظت، عزيزي الدارس، بأن الاختلاف بين القوائم الثلاثة هو في طريقة تنظيم العناصر. فالعناصر نفسها ثابتة، والسياق مختلف.

والآن بعد أن رأينا هذا المثال، نعود إلى السؤال الذي طرحناه قبل قليل فيما يتصل بإمكانية الصيغ السابقة على التعبير عن المشكلات المماثلة للمشكلة التي عرضناها في هذا المثال. والإجابة، كما ترى، هي أن هذه الصيغ محدودة الإمكانات، وبذلك فنحن بحاجة إلى نمط جديد من القوائم المتصلة للقيام بهذه المهمة. وهذا النمط هو ما نطلق عليه اسم القوائم المشتركة، وهي موضوع حديثنا هنا. والصيغة الممثلة في الشكل (46)، ما هي إلا مثال على هذا النوع من القوائم المتصلة.

وهذا النوع من القوائم المتصلة يتسم بصفتين أساسيتين، هما:

1. أن العنصر الواحد يشترك في أكثر من قائمة.
2. أن هناك على الأقل مؤشرين لكل عنصر، وقد يزيد العدد عن ذلك تبعاً لعدد القوائم، وحاجة المشكلة التي نحن بصدد التعبير عنها.

لاحظ أيضاً أن العمليات التي يمكن أن تجري على القوائم المشتركة أكثر تعقيداً من العمليات التي تم وضعها فيما يتصل بالأنواع الأخرى من القوائم المتصلة. ومرد ذلك ضرورة التعامل مع سياقات ومؤشرات متعددة. وكلما زاد عدد المؤشرات، زادت درجة تعقيد العمليات التي تجري على القوائم المشتركة. ومما يزيد الأمر سوءاً هو أن القوائم المشتركة لا تظهر بصيغة موحدة حتى يمكن أن نصف العمليات التي تتم عليها بخطوات محددة على النحو الذي رأيناه في الأنواع الأخرى من القوائم المتصلة. فاختلاف التطبيق يؤدي إلى اختلافات في الخوارزميات، وبصفة خاصة خوارزميات الإضافة والحذف.

افترض، على سبيل المثال، أننا نريد أن نقوم بعملية حذف لأحد عناصر القائمة المشتركة الممثلة في الشكل (46)، فإن ذلك يقتضي منا أن نقوم بتحديد مكان هذا العنصر (ولنقل المكان هو posn) وأن نقوم أيضاً بتحديد العنصر السابق له، على النحو الذي بيناه فيما سبق عند الحديث عن الأنواع الأخرى من القوائم المتصلة. ولكن في حالة هذه القائمة ليس لدينا عنصر سابق واحد، بل لدينا ثلاثة عناصر نتيجة لوجود ثلاثة سياقات (أو قل قوائم) مختلفة. ومعنى ذلك أننا بحاجة إلى تحديد مكان هذه العناصر الثلاثة.

فلو كان العنصر المراد حذفه هو ذلك العنصر الممثل بالقيمة "Jordan"، مثلاً، فإن المؤشرات الثلاثة الأخرى للعناصر السابقة ستكون على النحو التالي:

- مؤشر يشير إلى العنصر السابق الممثل بالقيمة "Baker" ضمن التسلسل الهجائي للأسماء.

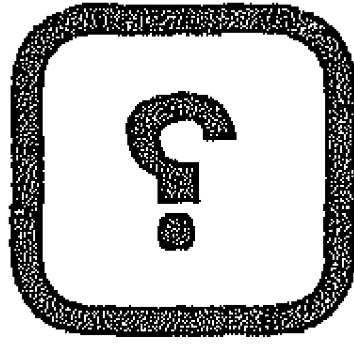
- مؤشر يشير إلى العنصر السابق الممثل بالقيمة "Parker" ضمن التسلسل الخاص بأرقام الحسابات.

- مؤشر يشير إلى العنصر السابق الممثل بالقيمة "Samwell" ضمن التسلسل الخاص بالأرقام العائلية.

ومعنى ذلك أنه ينبغي أن نقوم بالتعديل على هذه المؤشرات الثلاثة لتصبح في وضع تستطيع معه الإشارة إلى العنصر التالي للعنصر المحذوف ضمن السياقات الثلاثة المذكورة.

أما في حالة الإضافة، فإن الأمر يقتضي أن تقوم بفك ارتباط المؤشرات الثلاثة للعنصر السابق، إذا وقع العنصر الجديد بين عنصرين آخرين، وإعادة ربطها بالعنصر المضاف، وجعل العنصر الجديد يشير إلى ما كان يشير إليه العنصر السابق. وهذا يستلزم أيضاً التعامل مع ثلاثة مؤشرات.

أسئلة التقويم الذاتي (2)



1. يحتوي المؤشر على قيمة معينة نشير إليها بالمصطلح ....
2. التمثيل المتصل باستخدام المؤشرات يصلح بصفة خاصة في اللغات ....
3. هناك ثلاث طرق لتمثيل القوائم المتصلة في الذاكرة هي: ....
4. الهدف من وجود أماكن خالية في حالة المصفوفات المركبة أو المتوازية هو: ....
5. يستدل على بداية القائمة المتصلة باستخدام ....
6. في حالة المصفوفات المتوازية والمصفوفات المركبة ينبغي أن .... ونقوم بتعريف مصفوفة كافية لهذا العدد.
7. إن مسألة الاختيار بين الأساليب الثلاثة المذكورة لتمثيل القوائم المتصلة يتوقف على عاملين أساسيين هما: .... ، ....
8. عندما نقول قائمة متصلة مفردة فإننا نعني أن هذه القائمة تتخذ .... وقيمة آخر مؤشر فيها هي ....
9. السجل المكون لأي عنصر في القائمة المتصلة المفردة ينبغي أن يتضمن على الأقل حقلين أحدهما لـ .... والآخر لـ ....
10. يطلق على المؤشر الذي يشير إلى بداية القائمة المتصلة المصطلح ....
11. حتى تكون القائمة المتصلة دائرية ينبغي على مؤشر .... أن يشير إلى بداية القائمة.
12. من أمثلة عمليات الاستعراض التي تتم على القوائم المتصلة ما يلي: ....
13. إن عملية إضافة عنصر جديد إلى القائمة المتصلة يستلزم تحديد .... الذي ستتم فيه عملية الإضافة داخل القائمة.
14. تحدث الإضافة إلى بداية القائمة المتصلة المفردة في ثلاث حالات هي: ....، ....، ....

15. تحدث الإضافة في بداية القائمة المتصلة المفردة في حالتين هما: .... ، ....

16. في حالة حذف أحد العناصر من القائمة المتصلة المفردة فإننا نحتاج إلى مؤشرين أحدهما لـ .... والآخر لـ ....

17. هناك أربع صيغ يمكن أن تظهر بها القوائم الدائرية هي: .... ، .... ، .... ، ....

18. القائمة الدائرة الخالية (empty) تتكون من .... فقط ويكون المؤشر الخارجي في وضع يشير فيه إلى .... أما المؤشر الأخير فيشير أيضاً إلى ....

19. هناك أربع صيغ يمكن أن ترد بها القائمة المتصلة الثنائية هي: ....، ....، .... ، ....

20. حتى تكون القائمة المتصلة قائمة مشتركة فإنها ينبغي أن تحقق شرطين أساسيين هما: .... ، ....

21. من بين الأنواع المختلفة للقوائم المتصلة فإن .... هي أكثر هذه الأنواع تعقيداً.

22. إن عملية إيجاد مكان أحد العناصر داخل القائمة المتصلة يتطلب في المتوسط زيارة عدد من العناصر يساوي تقريباً ....

23. تمتاز القائمة المتصلة الثنائية على الأنواع الأخرى من القوائم المتصلة بميزتين أساسيتين هما: .... ، ....

24. تكون قيمة المؤشر الخارجي للقائمة المتصلة الخالية في حالة عدم وجود رأس للقائمة ....

6. المصفوفات الثنائية والمتعددة الأبعاد

كما هو الحال بالنسبة للمصفوفات ذات البعد الواحد، فإن لغات الحاسوب توفر ضمن إمكاناتها البرمجية ما يمكّن المبرمج من التعامل مع الجداول على اختلاف أشكالها وأبعادها، مع بعض التفاوت فيما بينهما من حيث عدد الأبعاد التي تسمح بها. فهناك بعض المشكلات التي تحتاج، بحكم طبيعتها، إلى عرض بياناتها بشكل جدولي أو على نحو يعكس أبعاداً جدولية متعددة. ومن هنا، فإن هذا النوع من التراكيب البيانية يتيح للمبرمج سهولة التعامل مع هذه البيانات سواء في التخزين أو الوصول والمعالجة.

ولكن المصفوفات المتعددة الأبعاد (ثنائية أو غير ذلك)، على خلاف المصفوفات الأحادية، لا تستلزم القيام بالعمليات ذاتها التي أشرنا إليها في الجزء السابق كعمليات الحذف والإضافة والفرز، مما يجعل التعامل معها مقصوراً على نطاق محدود من التطبيقات، إلا إذا نظر إلى المصفوفة الواحدة على أنها مجموعة مركبة من القوائم والمصفوفات الأحادية. وفي هذه الحالة، فإن العمليات التي سبق أن أشرنا إليها تصبح ضرورية.

ومن بين المسائل الأساسية التي تبرز لدى التعامل مع المصفوفات الثنائية والمتعددة، مسألة تحديد طريقة تمثيلها في الذاكرة وبالتالي تحديد العناوين المطلقة للعناصر. إلا أنه ينبغي التنبيه إلى أن هذه ليست مهمة المبرمج. فلغات البرمجة هي التي تعالج ذلك. وتسير هذه اللغات وفقاً لأحد أسلوبين: الأول هو التمثيل الطولي (column-major ordering) والآخر هو التمثيل الأفقي (row-major ordering). ولتبيان الفرق بينهما تأمل الشكل (48):

	1	2	3	4
1	11	12	13	14
2	21	22	23	24
3	31	32	33	34
4	41	42	43	44
5	51	52	53	54

الشكل (48): مصفوفة ذات بعدين.

فالقيم الممثلة في هذا الشكل يمكن أن ينظر إليها من زاويتين مختلفتين، وذلك على النحو الموضح في الشكل (49). ففي التمثيل الأفقي يتم تخزين القيم الواردة في الصف الأول متلوة بالقيم في الصف الثاني... وهكذا حتى الصف الأخير في الجدول. أما في التمثيل الطولي، فإن التخزين يسير وفق الأعمدة؛ إذ يتم أولاً تخزين القيم الواردة في العمود الأول متلوة بالقيم الواردة في العمود الثاني،... وهكذا حتى آخر عمود في الجدول.

أفقي	11	12	13	14	21	22	23	24	31	32	33	34	41	42	43	44	51	52	53	54
طولي	11	21	31	41	51	12	22	32	42	52	13	23	33	43	53	14	24	34	44	54

الشكل (49): أسلوب تمثيل المصفوفة الثنائية في الذاكرة.

وهذه المسألة لا تقتصر في الحقيقة على المصفوفات ذات البعدين فقط. ففكرة التمثيل الأفقي أو الطولي تنطبق على المصفوفات ذات الأبعاد المتعددة. ولتوضيح ذلك، دعنا نتأمل الشكل (50) الذي يعبر عن مصفوفة ذات ثلاثة أبعاد [3][4][3] A .

Page(3)

	1	2	3	4
	88	91	94	97
	89	92	95	98
	90	93	96	99

Page(2)

	44	47	50	53
	45	48	51	54
	46	49	52	55

Page (1)

1	11	14	17	20
2	12	15	18	21
3	13	16	19	22

الشكل (50): تمثيل طولي في الذاكرة لمصفوفة ذات ثلاثة أبعاد

```
int A[3] [4] [3];
```

ففي حالة التمثيل الطولي، تتخذ القيم السياق الموضح في الشكل (51)، حيث سرنا أولاً مع القيم الواردة في العمود الأول، ثم القيم الواردة في العمود الثاني، فالثالث... وهكذا حتى انتهينا من الصفحة الأولى، ثم بعد ذلك قيم العمود الأول في الصفحة الثانية، فالعمود الثاني... وهكذا حتى انتهت قيم الصفحة الثانية، ثم تتلو ذلك قيم الصفحة الثالثة، عموداً بعد عمود.

11	12	13	14	15	16	17	18	19	20	21	22	44	45	46	47	48	49	
	50	51	52	53	54	55	88	89	90	91	92	93	94	95	96	97	98	99

الشكل (51): تمثيل طولي في الذاكرة لمصفوفة ذات ثلاثة أبعاد.

أما في حالة التمثيل الأفقي، فإن القيم ستخزن في ذاكرة الحاسوب على النحو الموضح في الشكل (52). وكما هو واضح من هذا الشكل، فقد قمنا بتخزين القيمة الأولى في الصف الأول من الصفحة الأولى، وتلتها القيمة الأولى في الصف الأول من الصفحة الثانية، تلتها القيمة الأولى في الصف الأول من الصفحة الثالثة. ثم انتقلنا إلى القيمة الثانية في الصف الأول من الصفحة الأولى، وتلتها القيمة الثانية من الصفحة الثانية، وتلتها القيمة الثانية في الصف الأول من الصفحة الثالثة... وهكذا حتى انتهت قيم الصف الأول في الصفحات الثلاثة. وانتقلنا بعدها إلى الصف الثاني وقمنا بتخزين قيمه بالأسلوب نفسه، وهكذا حتى يتم تخزين جميع القيم.

11	44	88	14	47	91	17	50	94	20	53	97	12	45	89	15	48	92	
18	51	95	21	54	98	13	46	90	16	49	93	19	52	96	22	55	99	

الشكل (52): تمثيل أفقي في الذاكرة لمصفوفة ذات ثلاثة أبعاد.

ولو قارنا بين الشكل (51) والشكل (52)، لوجدنا أن هناك فرقاً كبيراً بين الاثنين. فلو نظرنا إلى القيمة المخزنة في الموضع الثامن في الشكل الأول لوجدنا أن هذه القيمة هي (18) ولو بحثنا عن مكان تخزين القيمة نفسها وفق التمثيل الأفقي، لوجدنا أنها مخزنة في الموضع التاسع عشر في الشكل (52).

وهنا يبرز السؤال المهم: كيف يمكن تحديد العنوان المطلق لعنصر معين داخل الذاكرة؟

وللإجابة على ذلك نقول: أننا بحاجة إلى إجراء بعض العمليات الحسابية للوصول إلى

ذلك. والمعلومات التي نحتاجها لهذه الغاية هي:

أ. العنوان المطلق للعنصر الأول في الذاكرة وهو ما نسميه الأساس (base).

ب. عدد المواضع التخزينية التي يحتلها العنصر الواحد.

ج. البعد الطولي للعنصر المعني بالنسبة إلى الأساس.

ويمكن التعبير عن ذلك بالمعادلة التالية:

$$\alpha(A[K_1, K_2, ..., K_n]) = base + size(offset)$$

ولعلك تذكر أن هذه المعادلة شبيهة بالمعادلة التي أوردناها عند حديثنا عن المصفوفات ذات البعد الواحد. وينعكس الفرق بين التمثيل الأفقي والتمثيل الطولي في طريقة حساب الجزء الأخير من المعادلة، وهو البعد الطولي (offset)، والذي يتم على النحو التالي:

أولاً: التمثيل الطولي (column-major)

$$offset = E_n L_{n-1} ... L_1 + E_{n-1} L_{n-2} ... L_1 + ... + E_2 L_1 + E_1$$

ثانياً: التمثيل الأفقي (row-major)

$$offset = E_1 L_2 ... L_n + E_2 L_3 ... L_n + ... + E_{n-1} L_n + E_n$$

وتُحسب كل من E و L على النحو التالي:

$$L_i = B' \;_i - B \;_i + 1$$
$$E_i = K_i - B$$

علماً بأن:

K_i: تشير إلى دالة نسبية (subscript)

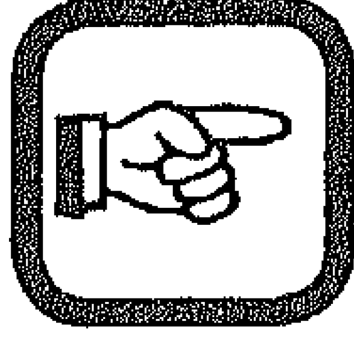
ومثال ذلك قولنا [3][2] A أي بمعنى A[K1][K2]

LB: القيمة الدنيا لكشاف المصفوفة (lower-bound)

كما ورد في التعريف، فإذا قلنا [10] array; فإن القيمة الدنيا في هذه الحالة هي "0".

UB: القيمة القصوى لكشاف المصفوفة (upper-bound) كما ورد في التعريف، وبذلك فإن القيمة القصوى في المثال الوارد أعلاه هي "9" .

والمثال التالي يوضح هذه الحسابات.



مثال (11)

إذا علمنا، في ضوء ما هو معطى في الشكل (49) أن: (base = 1001) و(LB =0) و(size =1) فما هو العنوان المطلق في الذاكرة للعنصر ؟ [2][3][1] A والإجابة هي كما يلي:

$$L_i = B_i - B_i + 1$$

$E_1 = 1 - 0 = 1$

$E_2 = 3 - 0 = 3$

$E_3 = 2 - 0 = 2$

$$E_i = K_i - B$$

$L_1 = 2 - 0 + 1 = 3$

$L_2 = 3 - 0 + 1 = 4$

$L_3 = 2 - 0 + 1 = 3$

أولاً: وفق التمثيل الطولي

$$\alpha(A[2,3,2\} = 1001 + 1[\ 2 * 4 * 3 + 3 * 3 + 1] = 1035$$

ثانياً: وفق التمثيل الأفقي

$$\alpha(A[2,2,3\} = 1001 + 1[\ 1 * 4 * 3 + 3 * 3 + 2] = 1024$$



أسئلة التقويم الذاتي (3)

اختر الإجابة المناسبة لكل عبارة من العبارات التالية:

1. يتم تثبيت عدد عناصر المصفوفة خلال:
   1. مرحلة الترجمة
   2. مرحلة تنفيذ البرنامج
   3. عند إسناد أول قيمة للمصفوفة
   4. عند الانتهاء من إسناد جميع القيم للمصفوفة
2. إن عدد القيم التي تضمها المصفوفة ذات البعد الواحد:
   1. يمكن أن يكون صفراً
   2. يمكن أن يكون مساوياً لعدد المواضع المحجوزة
   3. يمكن أن يزيد أو ينقص تبعاً للحاجة
   4. كل ما ذكر
3. تستغرق عملية تحديد العنصر الأخير في القائمة المخزنة في المصفوفة الأحادية:

1. وقتاً يتناسب مع عدد القيم
2. وقتاً يتناسب مع عدد المواضع المحجوزة للمصفوفة.

4. تستغرق عملية تحديد الموضع الأخير في المصفوفة الأحادية:
1. وقتاً ثابتاً 2. وقتاً يتناسب مع عدد القيم

5. تستغرق عملية حذف أحد عناصر القائمة المخزنة في مصفوفة أحادية وقتاً يتناسب مع:
1. عدد القيم المخزنة
2. عدد المواضع المحجوزة.
3. عدد القيم المزاحة باتجاه نهاية المصفوفة.
4. عدد القيم المزاحة باتجاه بداية القيمة.

6. تستغرق عملية إضافة أحد العناصر الجديدة إلى القائمة المخزنة في مصفوفة أحادية وقتاً يتناسب مع:
1. عدد القيم المخزنة
2. عدد المواضع المحجوزة
3. عدد القيم المزاحة باتجاه نهاية المصفوفة.
4. عدد القيم المزاحة باتجاه بداية القائمة.

7. إذا لم يكن باستطاعتنا تحديد الحد الأعلى لعدد القيم التي سنتعامل معها فإن من الأفضل اختيار
1. القوائم المتصلة المفردة التي تستخدم المؤشرات.
2. القوائم الملتزة الممثلة بالمصفوفات الأحادية.
3. القوائم المتصلة الممثلة على شكل مصفوفات متوازية.
4. القوائم المتصلة الممثلة على شكل مصفوفات مركبة.

8. يبلغ متوسط عدد المقارنات لإيجاد موقع إحدى القيم في المصفوفة
1. نصف عدد عناصر المصفوفة تقريباً.
2. العدد الكلي لعناصر المصفوفة.
3. نصف عدد القيم المخزنة في المصفوفة تقريباً.
4. العدد الكلي للقيم المخزنة.

9. إذا كان لدينا التعريف التالي (int A[3][5];) وإذا علمنا أن عنوان الأساس هو 2001، وان العنصر الواحد يحتل بايتاً واحداً في الذاكرة، فإن العنوان المطلق للعنصر A[2][3] باستخدام أسلوب التخزين الطولي (column-major) هو:
1. 2007 2. 2008 3. 2009 4. لا شيء مما ذكر

10. استخدم المعلومات المعطاة أعلاه لتحديد العنوان المطلق للعنصر A[2][5] باستخدام أسلوب التخزين الأفقي (row-major):
1. 2014 2. 2010 3. 2008 4. لا شيء مما ذكر

11. إذا كان لدينا التعريف التالي (int [3][3][2]) وإذا علمنا أن عنوان الأساس هو 2001، وان العنصر الواحد يحتل موضعاً واحداً في الذاكرة، فإن العنوان المطلق للعنصر A[2][1][2]باستخدام التمثيل الطولي هو:

1. 2001   2. 2010   3. 2014   4. لا شيء مما ذكر

12. يتم تخزين المصفوفات المتعددة الأبعاد تتابعياً في الذاكرة باستخدام:

1. التمثيل الطولي   2. التمثيل الأفقي   3. التمثيل الطولي والأفقي

13. لكي يتم استعراض جميع عناصر المصفوفة ذات الأبعاد الثلاثة فإننا نحتاج إلى تركيب دوراني مكون من:

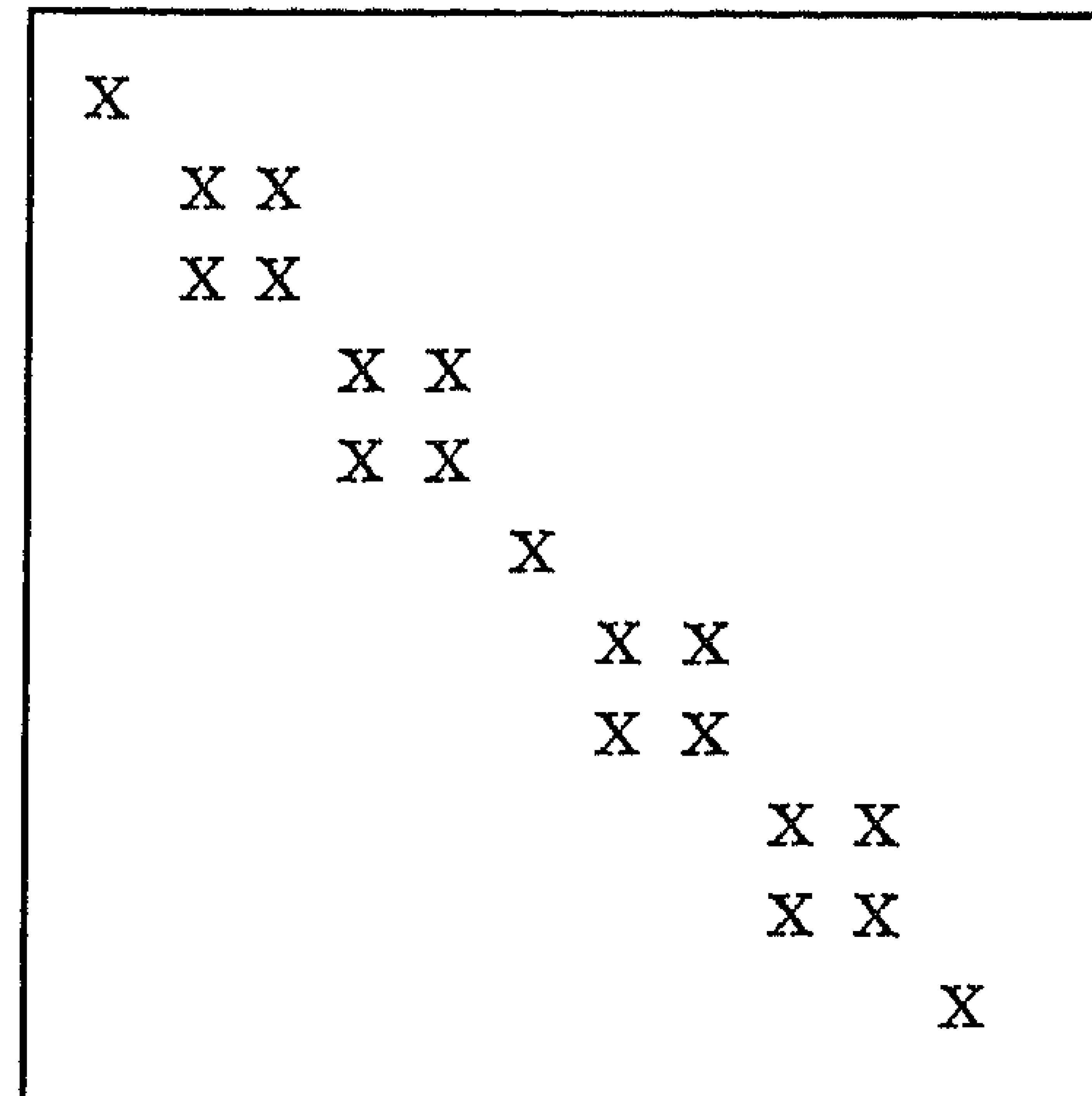
1. (4   2. (for I: =...)   3. (for I: =...)   (for I: = ...) .لا شيء مما ذكر

(for J)   (for J: = ...)

(for K: = ...)

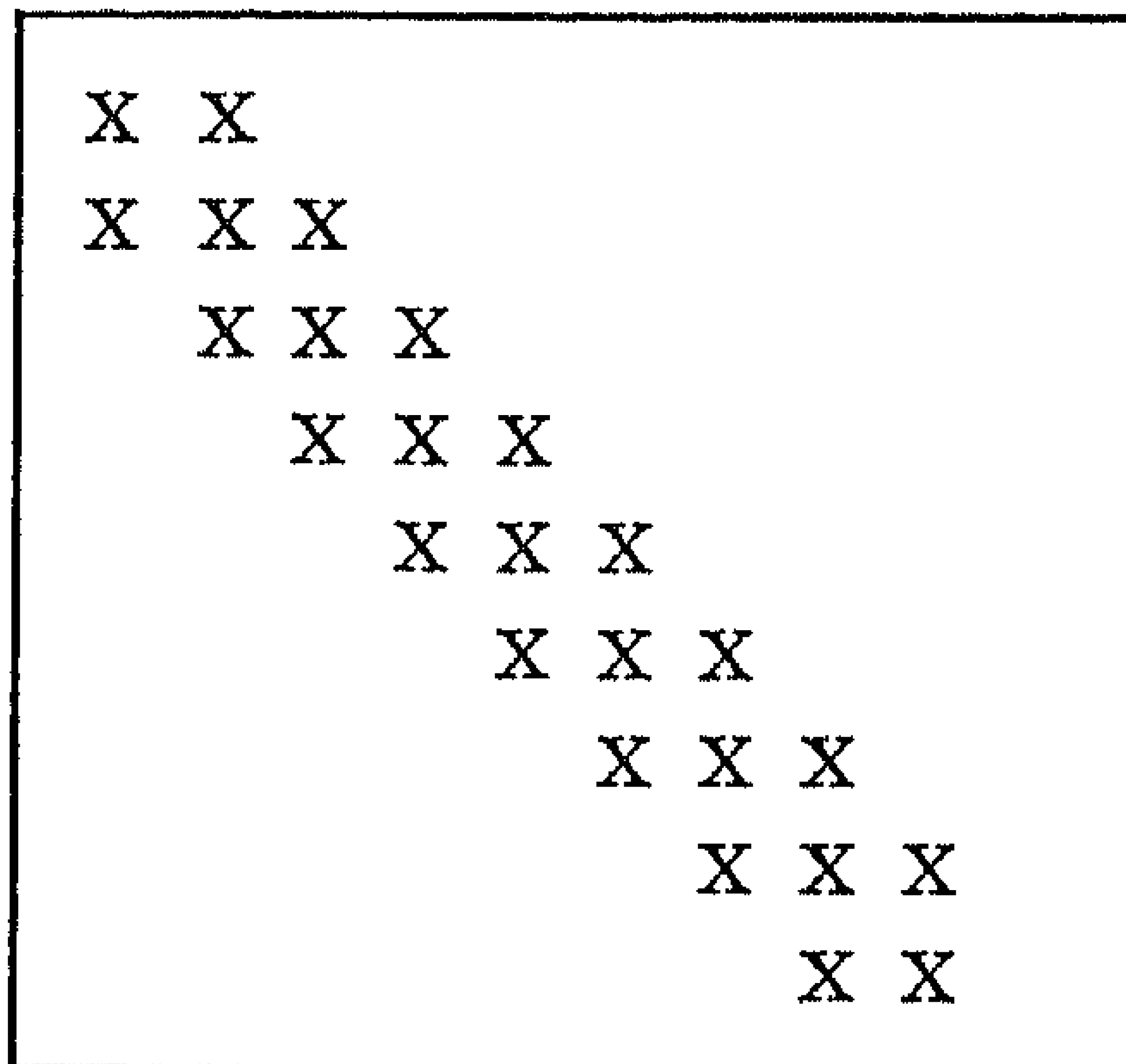
الجداول الشتيتة (sparse matrices)

لقد تحدثنا قبل قليل عن المصفوفات المتعددة الأبعاد، بما في ذلك المصفوفات الثنائية. وهنا ينبغي الإشارة إلى أن هناك بعض الحالات التي لا تحتاج فيها كل خلية في المصفوفة الثنائية إلى وجود قيمة محدّدة. وهناك بعض الحالات التي تستلزم وجود قيمة الصفر في بعض خلايا المصفوفة والشكل (53) يعطي نماذج من المصفوفات الثنائية ذات القيم المتفرقة.



الجدول القطري الرباعي

(block diagonal)



الجدول القطري الثلاثي

(tridiagonal)

X	X	X	X	X	X	X	X
	X	X	X	X	X	X	X
		X	X	X	X	X	X
			X	X	X	X	X
				X	X	X	X
					X	X	X
						X	X
							X

الجدول المثلث (العلوي)

(upper triangular)

X							
X	X						
X	X	X					
X	X	X	X				
X	X	X	X	X			
X	X	X	X	X	X		
X	X	X	X	X	X	X	
X	X	X	X	X	X	X	X

الجدول المثلث (السفلي)

(lower triangular)

الشكل (53): بعض الصيغ التي ترد بها الجداول الشتيتة

ومن بين الصيغ المختلفة للمصفوفة الممثلة في الشكل (53)، سنتناول بالحديث الصيغة المثلثة ذات القاعدة السفلية (lower triangular). وكما تلاحظ، فإن الصفر أو الفراغ في هذه المصفوفة يحتل كل موضع قيمة دالته العمودية (column-subscript) أكبر من قيمة دالته الأفقية (row-subscript)، فهو يحتل جميع مواضع الصف الأول فيما عدا الموضع الأول (من اليسار)، ويحتل مواضع الصف الثاني باستثناء الموضعين الأول والثاني (من اليسار)... وهكذا حتى الصف الأخير حيث لا يحتل الصفر أو الفراغ أية خلية.

ولتبسيط عملية التعامل مع هذا الجدول ولتجنب إهدار مساحة التخزين المتوفرة، فإن من الممكن تمثيله على شكل مصفوفة أحادية. وبذلك لو كانت لدينا المصفوفة المثلثة الموضحة في الشكل (54-أ) فإن من الممكن تمثيلها في الذاكرة بطريقة تتابعية على النحو التالي الموضح في الشكل (54 – ب).

أ - جدول مثلث بوضعه المنطقي

J

I						
	a					
	i	f				
I	t	h	e			
	f	o	u	r		
	l	o	w	e	r	
	a	r	r	a	y	s

ب - التمثيل التتابعي للجدول المثلث

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	i	f	t	h	e	f	o	u	r	l	o	w	e	r	a	r	r	a	y	s

الشكل (54): أسلوب تمثيل المصفوفة المثلثة في الذاكرة

والسؤال الذي يبرز الآن هو: كيف يمكن أن نحدد عنوان العنصر الواحد وبنفس الوقت نحافظ على التصور المنطقي للجدول الشتيت؟

للإجابة على هذا التساؤل نقول: إن هناك طريقتين: الأولى حسابية والأخرى باستخدام كشاف خاص لتحديد المواضع. فالطريقة الحسابية تقوم على المعادلة التالية:

$$A[I][J] = I(I-1) / 2 + (J-1)$$

ومعنى ذلك، أننا لو أردنا معرفة موضع تخزين العنصر A[4][2] المتمثل بالحرف "O" فإن ذلك يمكن أن يحتسب كما يلي:

$$A[4][2] = 4(4-1) / 2 + (2-1) = 7$$

لو حاولنا أن نتأكد من ذلك من خلال التمثيل الموضح في الشكل (54) لوجدنا أن هذا الحرف يحتل فعلاً الموضع رقم (8) الذي وصلنا إليه عن طريق المعادلة أعلاه.

أما طريقة استخدام الكشاف (ACCESS INDEX) فيمكن تنفيذها على النحو الموضح في الشكل (55). فكما تلاحظ أن الكشاف يضم إشارات إلى القيم الأولى في

مصفوفة J

كشاف	I						
0		a					
1		i	f				
3	I	t	h	e			
6		f	o	u	r		
10		l	o	w	e	r	
15		a	r	r	a	y	s

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	i	f	t	h	e	f	o	u	r	l	o	w	e	r	a	r	r	a	y	s

الشكل (55): استخدام كشاف لتحديد مواضع تخزين العناصر

الصفوف الستة للجدول. وبذلك لو أردنا أن نصل إلى العنصر A[4][2] فإننا سنبحث في الكشاف عن مكان تخزين العنصر الأول في الصف الرابع وهو الموضع رقم (6) في التمثيل التتابعي. ومن تلك النقطة نستطيع الوصول إلى العنوان المطلوب بتطبيق العلاقة الحسابية التالية:

$$A[4][2] = \text{ROW access value} + (\text{column subscript} - 1) = 6 + (2-1) = 7$$

وهذا هو العنوان نفسه الذي توصلنا إليه من خلال الطريقة الحسابية الأولى، لكن الطريقة السابقة تمتاز على هذه الطريقة من الناحية التخزينية؛ إذ أن هناك مساحة

إضافية مستهلكة لأغراض الكشاف في الطريقة الثانية. إلا أننا في المقابل يمكن أن نصل إلى عناصر كل صف على التتابع باستخدام الطريقة الثانية، وهو ما لا يتوفر في الطريقة الأولى إلا بإجراء حسابات لجميع عناصر الصف الأول.

هذا هو الأمر بالنسبة لتمثيل الجدول المثلث باستخدام أسلوب التمثيل التتابعي. وهذا الأسلوب، كما ترى، يتمتع بالبساطة وبالكفاءة أيضاً من حيث التخزين. لكن لاحظ أننا في حالة الرغبة في الوصول إلى العناصر الخاصة بكل صف أو كل عمود باستخدام الطريقة الأولى فإننا نضطر إلى القيام بحسابات تفصيلية مع الافتراض أيضاً بأننا ينبغي أن نكون على علم مسبق بمواضع القيم التي تم تخزينها، وهو ما لا يتيسر دائماً. وان تيسر فإنه يفرض على المبرمج قيوداً يصعب التحكم بها دائماً. وقد لاحظنا أيضاً بالنسبة للطريقة الثانية أنها لا تتيح إمكانية الوصول التتابعي للقيم الخاصة بالأعمدة كلٍ على حدة، شأنها في ذلك شأن الطريقة الأولى.

ولحل هذا الإشكال يمكننا أن نختار طريقة التمثيل المتصل (linked representation) باستخدام المؤشرات. والصيغة المناسبة لذلك هي القائمة المشتركة. وينبغي أن ننبه إلى أن هذه الطريقة، رغم أنها تزيد من مرونة الوصول وكفاءته فإنها تزيد من كلفة المساحة التخزينية وتزيد من درجة تعقيد الخوارزميات التي تنطوي عليها. ولبيان كيفية القيام بهذا النوع من التمثيل، دعنا نتأمل التعريف التالي.

```
class     nodeptr
 {
   struct noderec
      {int row, col;
             char info;
             nodeptr*  nextrow;
         nodeptr*  nextcol;
      };
   }sparse,ptr;
```

ويعبر هذا التعريف عن سجل مكون من خمسة حقول موزعة على النحو التالي، كما هو موضح في الشكل (56).

أ. القيمة المخزنة (info) (قيمة رمزية في الحالة التي نحن بصددها).

ب. رقم الصف الذي خزنت فيه القيمة (row).

ج. رقم العمود الذي خزنت فيه القيمة (col).

د. مؤشر إلى القيمة التالية في العمود (nextrow).

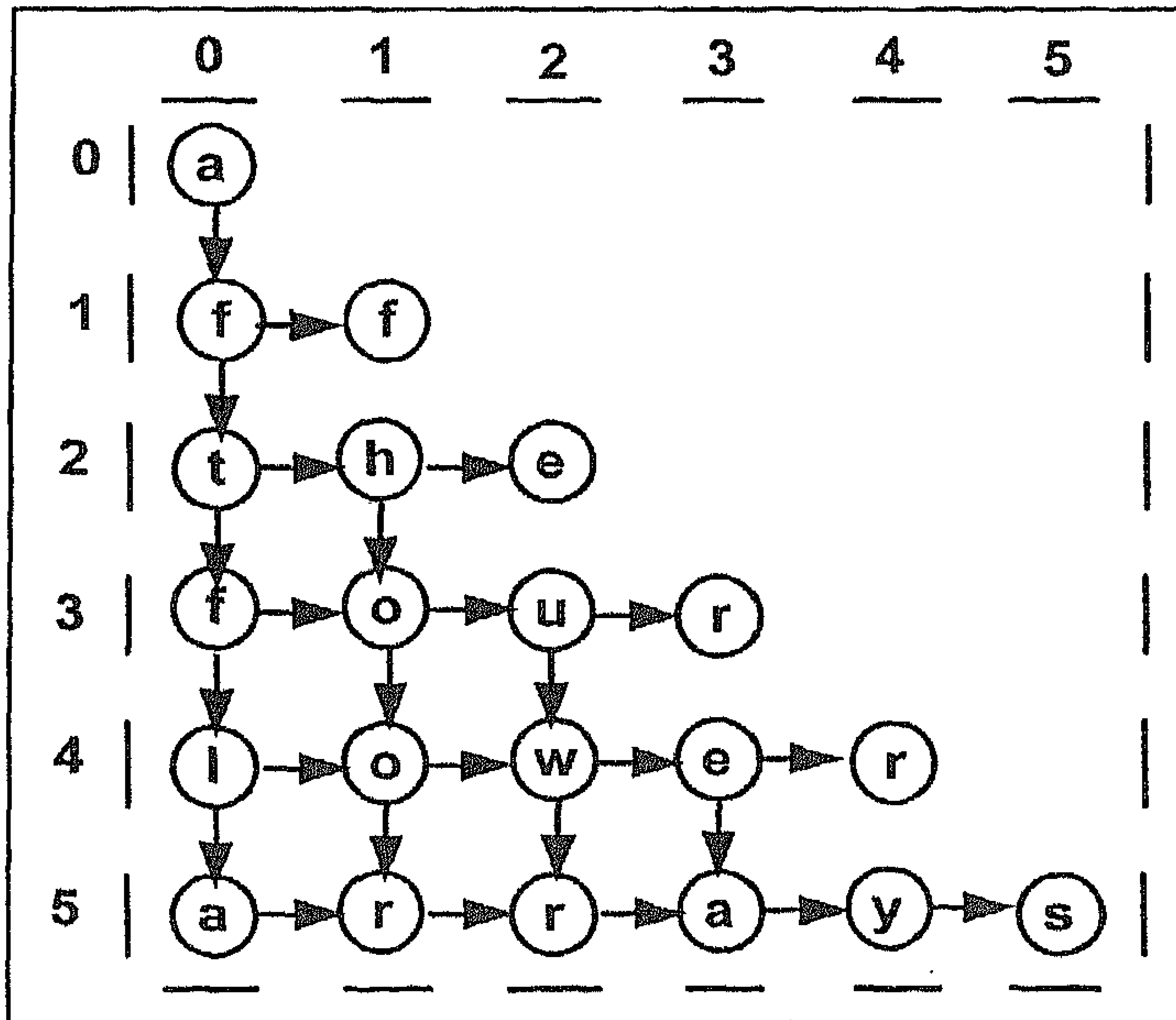
هـ. مؤشر إلى القيمة التالية في الصف (Nextcol).

row	*row*	*info*
nextrow		*nextcol*

الشكل (56): تمثيل لصيغة تعريف المتغير (ptr)

وباستخدام هذا السجل نستطيع أن نعبر عن جميع القيم التي تضمها المصفوفة المثلثة. فكل قيمة من هذه القيم سيكون لها سجل من هذا النوع. والسؤال المهم هو: كيف سنقوم بربط هذه العناصر معاً؟

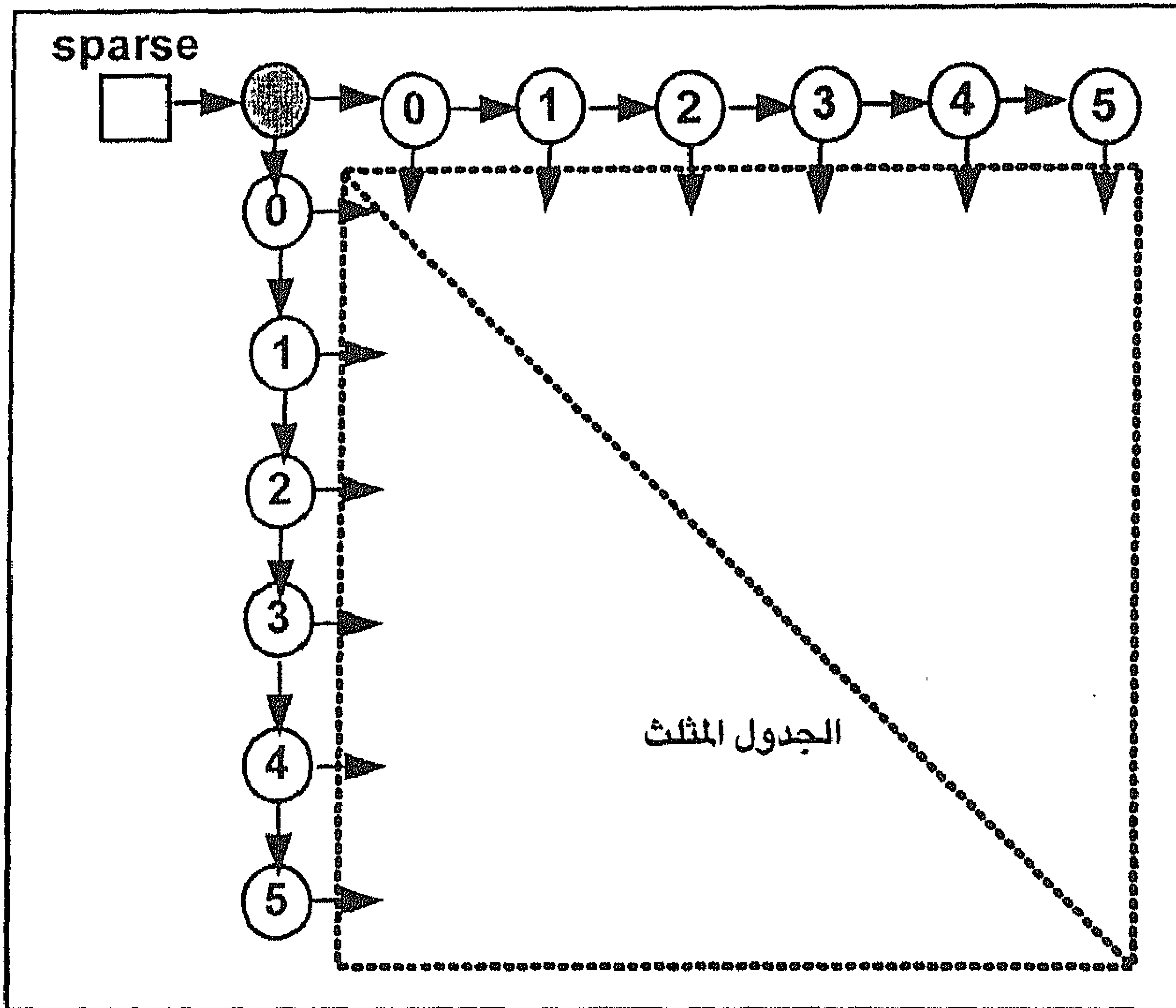
السبيل إلى ذلك هو استخدام المؤشرين (nextrow) و(nextcol). فكل عنصر يشير إلى ما يليه في الصف وفي العمود، على النحو الممثل في الشكل (57).



الشكل (57): مقطع من التمثيل المتصل للمصفوفة المثلثة.

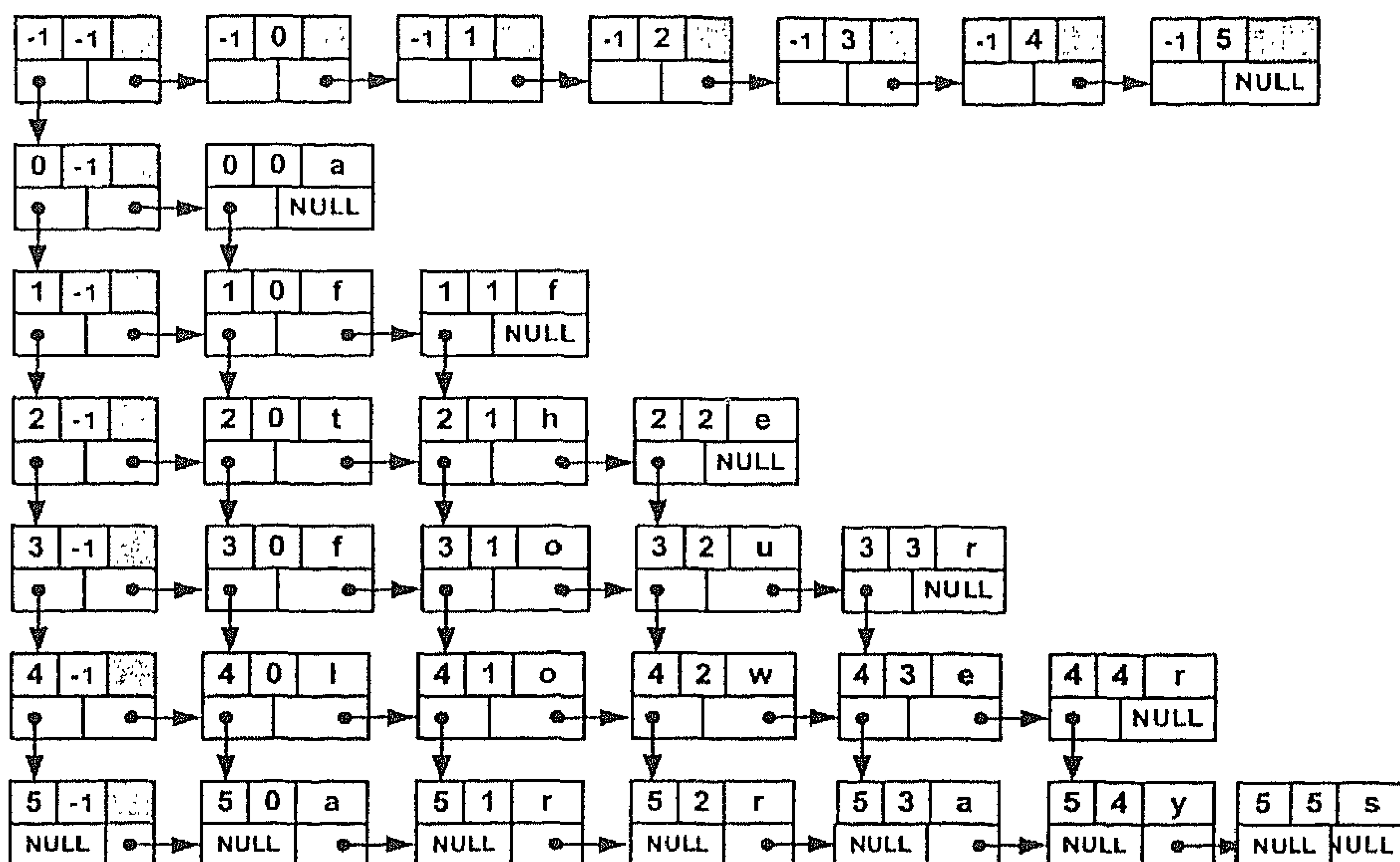
ولكن هذا التمثيل غير كافٍ بحد ذاته لتوفير إمكانية الوصول إلى الأعمدة والصفوف. فلا بد إذن من وسيلة مناسبة للإشارة إلى كل صف وكل عمود. وهذه الوسيلة هي إيجاد مؤشر لبداية كل قائمة من القوائم الإثني عشر الموجودة (وهي مجموع الصفوف والأعمدة). وبذلك يكون لدينا اثنا عشر مؤشراً: ستة منها للقوائم الأفقية والستة الأخرى للقوائم الطولية. بدلاً من أن يكون لكل مؤشر من هذه المؤشرات اسم مستقل وهو عدد

يصعب حصره والإحاطة به. وحتى تتوفر المرونة الكافية للانتقال من صف إلى آخر فإننا بحاجة إلى ربط هذه المؤشرات ببعضها بطريقة مناسبة. وهذه الطريقة هي تشكيل قائمة أفقية من المؤشرات، وأخرى طولية والربط بينهما بعنصر وهمي (dummy) على النحو المبين في الشكل (58). والحقيقة أن هذه المؤشرات ليست مؤشرات خارجية فحسب، بل هي سجلات من النوع المعرّف أعلاه، بذلك فهي تتضمن بعض المعلومات الخاصة، وهي على وجه التحديد رقم الصف أو العمود الذي تمثله. وبذلك فهي تعمل كمقدمات أو رؤوس (headers) للقوائم المختلفة.

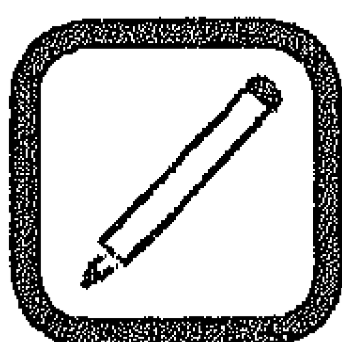


الشكل (58): سلسلة رؤوس القوائم الأفقية والقوائم العمودية للجدول المثلث

وبإيجاد هذه السلسلة من الرؤوس، ومؤشر خارجي للإشارة إلى التركيب البياني بأكمله فإننا نصل إلى تمثيل متكامل للجدول المثلث باستخدام فكرة القوائم المشتركة، وذلك على النحو الموضح في الشكل (59)، الذي يجمع في ثناياه ما ورد في الشكلين (57) و (58). بذلك، فإن بإمكاننا أن نصل إلى أي عنصر في المصفوفة من خلال العمود أو من خلال الصف، عبر المؤشر الخارجي sparse وعبر سلسلة المقدمات الأفقية أو المقدمات العمودية.



الشكل (59): تمثيل مصفوفة مثلثة باستخدام القوائم المشتركة



تدريب (15)

تأمل الجدول القطري الثلاثي (tridiagonal) الموضح في الشكل (60) واستخدم الأفكار التي درستها حتى الآن في هذا الجزء لبيان كيفية:

أ. تمثيل هذا الجدول بطريقة تتابعية وإيجاد عنوان العنصر[4][4] A

- أولاً باستخدام المعادلة التالية: $(A[I][J]) = 2I + J$

- ثانياً باستخدام فكرة الكشاف باستخدام المعادلة:

$$(A[I][J]) = \text{index value} + (\text{position} - 1)$$

ب. تمثيل هذه المصفوفة بطريقة متصلة على النحو المبين في الشكل (59).

A	B					
	C	D	E			
	F	G	H			
	·	I	J	K		
			L	M	N	
				O	P	Q
					R	S

الشكل (60): جدول قطري خاص بالتدريب (15)

7. الخلاصة

تطرقنا في هذه الوحدة، عزيزي الدارس، إلى بحث موضوع القوائم، فأشرنا إلى القائمة كتركيبة بيانات تجريدية. كما شرحنا لك كيفية تمثيل القوائم وتنفيذ عملياتها باستخدام القوائم المتصلة، والقوائم المفردة، والمصفوفات الأحادية. كما عرضنا موضوع محاكاة القوائم المتصلة باستخدام المصفوفات، وبينا لك أنواعاً أخرى من القوائم المتصلة مثل القوائم الدائرية، والثنائية، والمشتركة، كما أشرنا إلى المصفوفات الثنائية والمتعددة الأبعاد.

8. لمحة عن الوحدة الدراسية الخامسة

سنقوم عزيزي الدارس، في الوحدة التالية وهي الخامسة ببحث موضوع السلاسل الرمزية من حيث مفهومها وتعريفها وبناؤها باستخدام المصفوفة والمؤشرات. كما نقدم لك تطبيقات وأمثلة على استخدام السلاسل الرمزية.

9. إجابات التدريبات

تدريب (1)

```
void slist::maxim(slist* list, int& max)
    {slist* current;
  max =0;
  current=list;
  while (current != NULL)
   {if (max<current->info)
     max=current->info;
     current=current->link;
    }
  }
```

هذا بالنسبة لمعرفة القيمة العليا، أما بالنسبة للقيمة الدنيا بين القيم المخزنة في القائمة فإن الخوارزمية المذكورة أعلاه تستطيع أن تؤدي هذه المهمة بعد إجراء تغيير بسيط وهو:

```
if (min>current->info)
   min=current->info;
```

وكما تلاحظ فقد غيرنا اسم المتغير واستعملنا min بدلاً عن max كما يفضل تغيير اسم الدالة إلى minim بدلاً عن maxim .

تدريب (2)

1. مرتان   2. مرة واحدة   3. مرة واحدة

تدريب (3)

1. مرتان   2. مرة واحدة   3. مرة واحدة

تدريب (4)

```
void MtList::mean(MtList list,int first,int last)
  {int  sum=0, num=0, avrg=0 ;
   for (int i=first; i<last;i++)
   {sum += list.elements[i];
    num++;
   }
   avrg = sum/num;
   cout<<"mean:  "<< avrg;

  }
```

تدريب (5)

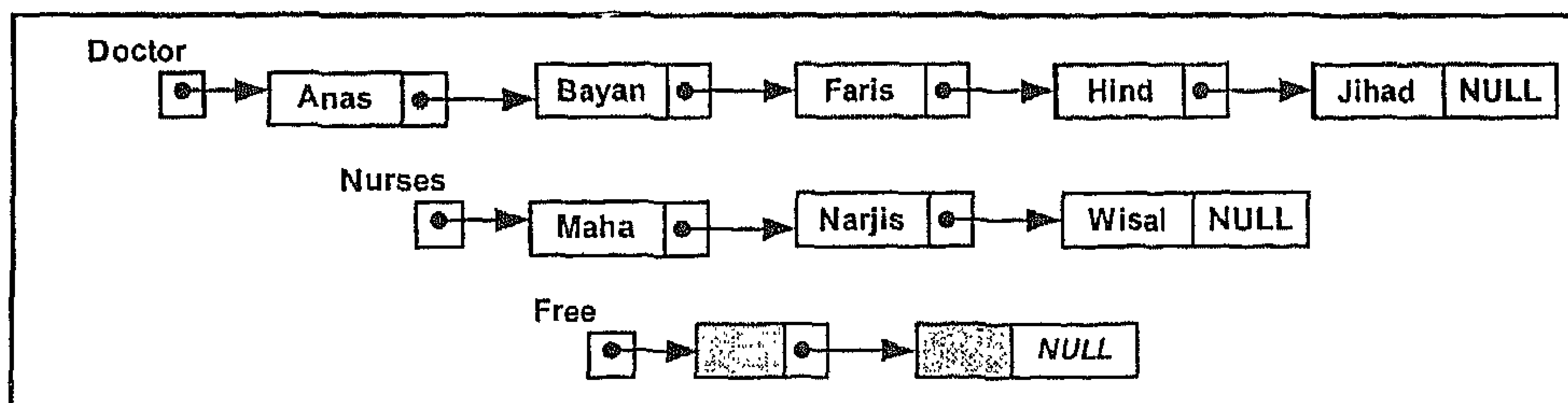
أ. ثماني مرات   ب. تسع مرات   ج. لا شيء

تدريب (6)

أ. لا شيء   ب. مرة واحدة   ج. ثماني مرات

تدريب (7)

أ. القوائم المتصلة الممثلة في الشكل (27)



ب. باستخدام المصفوفات المتوازية

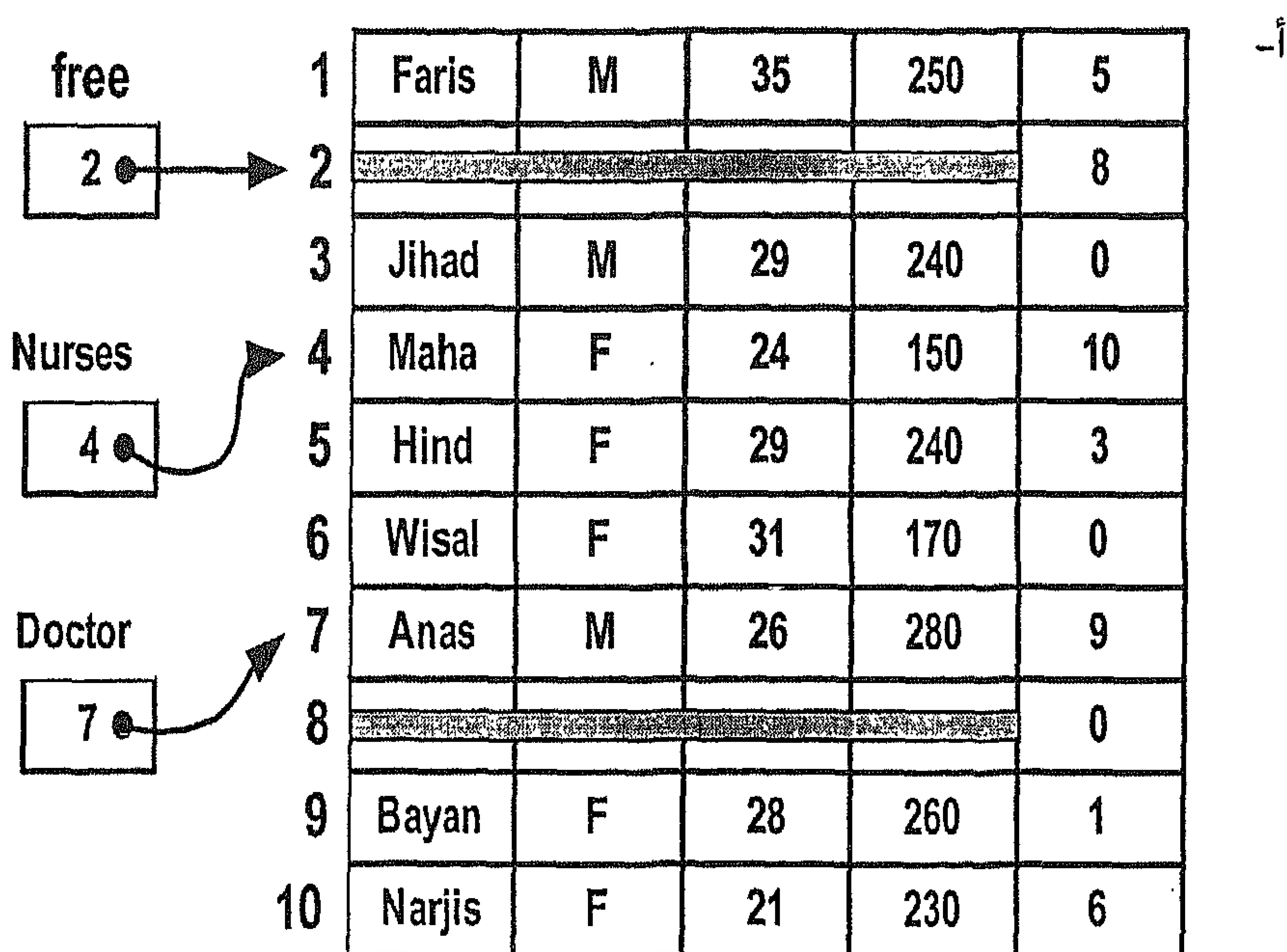
```
typedef char St[20];
  St name[10];
  char sex[10];
  int age[10];
  int sal[10] ;
  int link[10];
  int Doctor, Nurses, Free ;
```

ج. باستخدام المؤشرات

```
typedef char St[20];
 class  nodeptr
 {struct {
  St  name;
  char sex;
  int age;
  int sal;
  nodeptr * link;
  };
 }Doctor, Nurses, Free ;
```

تدريب (8)

أ-



1	Faris	M	35	250	5
2					8
3	Jihad	M	29	240	0
4	Maha	F	24	150	10
5	Hind	F	29	240	3
6	Wisal	F	31	170	0
7	Anas	M	26	280	9
8					0
9	Bayan	F	28	260	1
10	Narjis	F	21	230	6

ب-

```
struct noderec
  { char name[20];
    char sex;
    int age,sal,link ;
  };
 noderec list[10];
 int Doctor, Nurses, Free ;
```

تدريب (9)

```
void crList::traverseCl(crList * list)
  {
   while (list->info!=10)
     { process (list->info);
       list= list->link;
     }


  }
```

تدريب (10)

1. ثلاث مرات 2. مرتان 3. مرة واحدة

تدريب (11)

1. ثلاث مرات 2. مرة واحدة 3. مرة واحدة

تدريب (12)

أولاً: ستبقى كل من الخوارزميتين العاشرة (10) والثانية عشرة (12) بالوضع المحدد لهما ولا تغيير عليهما.

ثانياً: سيجري تغيير بسيط على كل من الخوارزميتين الحادية عشرة (11) والثالثة عشرة (13). ويتلخص هذا التغيير بما يلي:

أ. locatePos

```
while (next != list ->link && !found)
```

ب. findPos

```
while (current != list->link && current->info !=element)
```

تدريب (13)

أ. جملتا الاسـناد التاليتين:

```
list=posn->forw;
list->backw=NULL;
```

ب. جملتا الإسناد التاليتين:

```
list=posn->backw;
List->forw=NULL;
```

جـ. جملتا الإسناد التاليين:

```
Posn->backw->forw=posn->forw;
Posn->forw->backw=posn->backw;
```

تدريب (14)

.1

```
void dlList::insertDl(dlList* listt,dlList * pos, int element)
    { dlList *posn,*list, *newNode=new dlList;
     list=listt;
     posn=pos;
     newNode->info=element;
     posn=locatePos(list,posn, element);
     newNode->forw = posn->forw;
     posn-> forw= newNode;
     newNode->forw->backw = newNode;
     newNode->backw = posn;
    *listt= *list;
    *pos= *posn;
     }
```

.2

```
dlList * dlList::locatePos(dlList * list,dlList * posn,int element)
    {//finding a position for insertion
     dlList *previous, *next,*pos;
     bool found;
    previous = list;
    next=list->forw;
    found = false;
    while (next != list && !found)
     {if (element < next-> info)
       {posn=previous;
        found = true;
       }
       else
```

```
        {previous = next;
         next = next->forw;
        }
       pos = previous;
      }
     return pos;

    }
```

.3

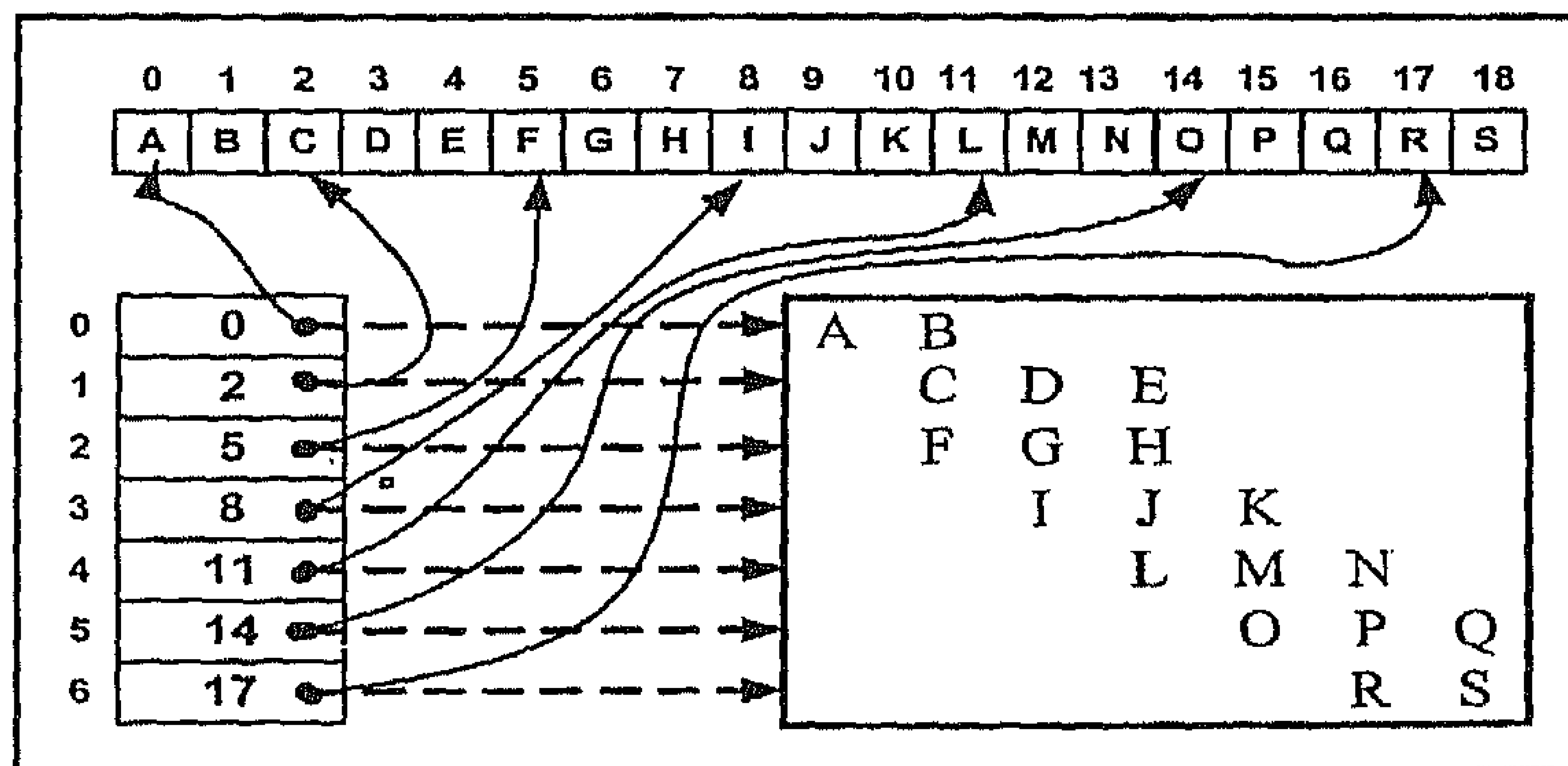
```
void dlList::deleteDl(dlList* listt , int element)
  {dlList *posn, *list=listt;
   posn=findPosn (list,posn, element);
   if (posn!= list)
     {posn->backw->forw=posn->forw;
      posn->forw->backw=posn->backw;
      delete(posn);
     }
   *listt=*list;
  }
```

.4

```
dlList* dlList::findPosn(dlList* list, dlList* posn,int element )
 {dlList *current;
     current=list->forw;
  while (current !=list && current->info !=element)
   current= current->forw;
   posn=current;
   return posn;
  }
```

تدريب (15)

أ- طريقة التمثيل في الذاكرة



العنوان المطلق: أولاً: $A [4][4]) = 2 I + J)$

$=8 + 4$

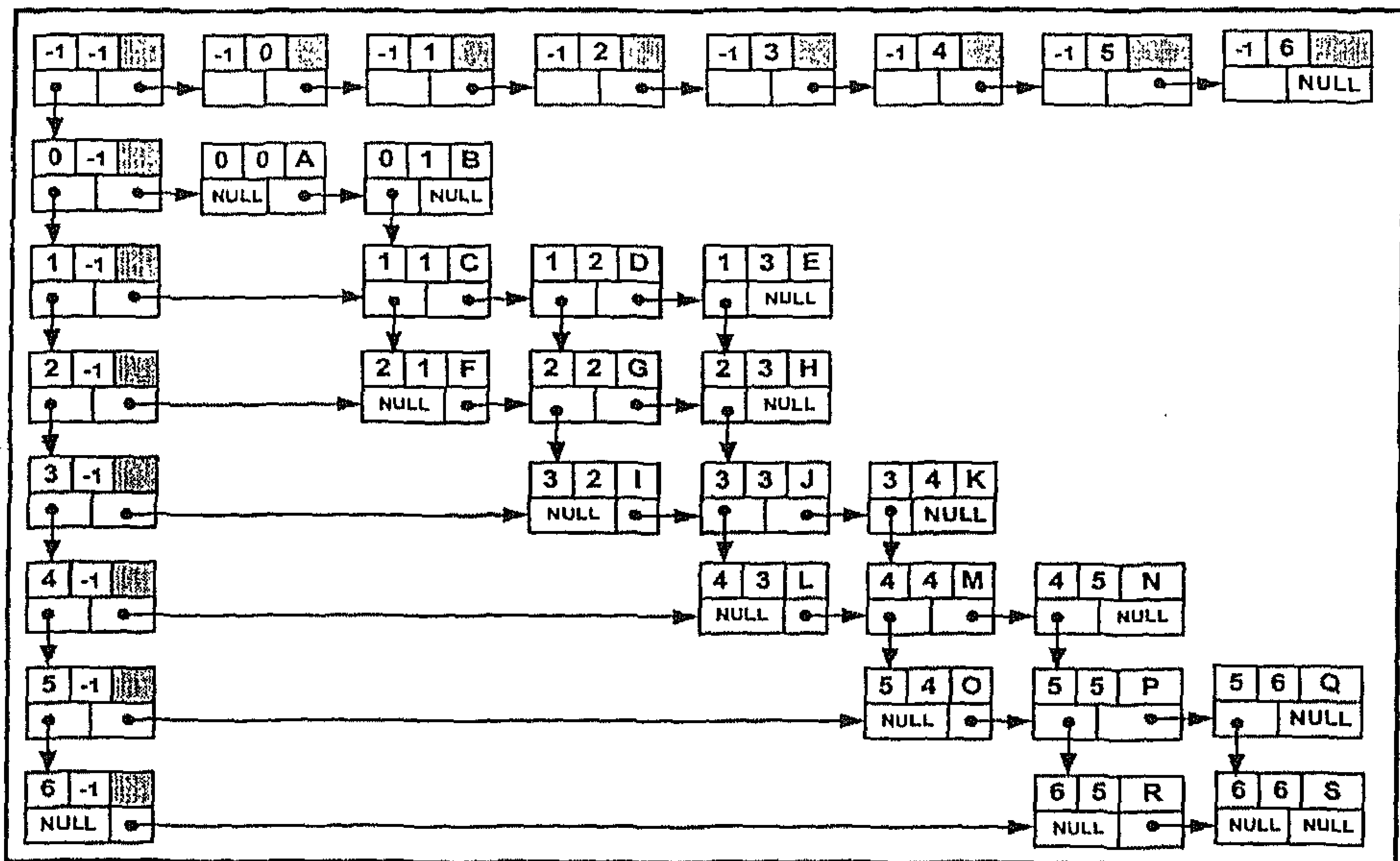
$=12$

ثانياً: $(A [4][4]) = \text{index value} + (\text{position} - 1)$

$=11+2-1$

$=12$

ب. تمثيل هذه المصفوفة بطريقة متصلة على النحو المبين في الشكل (58).



10. مسرد المصطلحات

- **الجدول الشتيت (sparse matrix)**: صيغة جدولية يشكل الصفر أو الفراغ جزءاً كبيراً منها. وقد ترد هذه الصيغة بأشكال مختلفة. فقد تكون قطرية، وقد تكون مثلثة، وقد تكون غير ذلك.

- **الحجز التلقائي للذاكرة (dynamic memory allocation)**: أسلوب خاص توفره بعض لغات البرمجة يتيح للمبرمج إمكانية حجز أماكن في الذاكرة حسب الحاجة وذلك خلال مرحلة التنفيذ.

- **الدالة النسبية (cursors)**: صيغة خاصة للمؤشرات تستخدم في المصفوفات المتوازية والمصفوفات المركبة في حالة تمثيل القوائم المتصلة. وهي تظهر على شكل قيمة صحيحة تحدد علاقة المواضع ببعضها وتسلسلها المنطقي في القائمة.

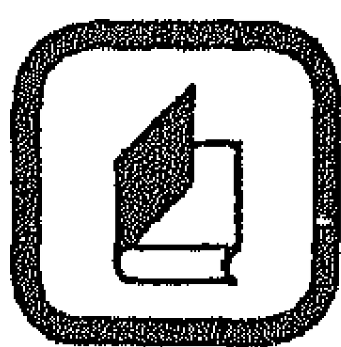
- **الذيل (trailer node)**: عنصر خاص يقع في نهاية بعض صيغ القوائم المتصلة. وقد يستخدمه المبرمج لتخزين بعض المعلومات الخاصة عن القائمة.

- **الرأس (header node)**: عنصر خاص يقع في بداية بعض صيغ القوائم المتصلة. وتخزن فيه بعض المعلومات العامة أو الخاصة عن القائمة، وقد يبقى خالياً من القيم.

- **القائمة الخالية (empty list)**: القائمة التي لا تضم أية قيم، بصرف النظر عن طريقة تمثيلها في الذاكرة.

- **القائمة الملتزة (dense list)**: مجموعة من العناصر تتخذ شكلاً خطياً وتخزن في مصفوفة ذات بعد واحد.

- **المؤشر الخارجي (external pointer)**: متغير مستقل يقوم بالإشارة إلى بداية القائمة المتصلة أو أي عنصر آخر فيها.



11. المراجع

1. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice- Hall. 2000.
2. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison Wesley, 1999.
3. Aho, Alfred; Hopcroft, John; Ullman, Jeffrey, Data Structures and Algorithms. Reading (USA): Addison-Wesley, 1983.
4. Kruse, Robert L., Data Structures and Program Design. Englewood Cliffs (USA): Prentice-Hall, 1984.
5. Lewis, T. G.; Smith, M.Z., Applying Data Structures. Atlanta (USA): Houghton Mifflin, 1976.
6. Lipschutz, Seymour, Schaum's Outline of Theory and Problems of Data Structures. New York: Mc Graw-Hill, 1986.
7. Tenenbaum, Aarom M.; Augenstein, Moshe J., Data Structures using Pascal. Englewood Cliffs (USA): Prentice-Hall, 1981.

الوحدة
الخامسة

السلاسل الرمزية Strings

محتويات الوحدة

1. المقدمة

1.1 تمهيد

أهلاً بك، عزيزي الدارس، في الوحدة الخامسة من كتاب "تركيب البيانات وتصميم الخوارزميات"، وهي بعنوان "السلاسل الرمزية"، والتي ستتعرف من خلالها على مفهوم السلاسل الرمزية (Strings)، وهي عبارة عن سلسلة من الرموز مثل 'Ali' و'ABC<>12'. وهو موضوع مهم لكونه يمثل تطبيقاً جيداً لما تعلمناه في الوحدات السابقة عن التراكيب التجريدية والقوائم المتصلة، وما إلى ذلك.

2.1 أهداف الوحدة

ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1. تعرف السلاسل الرمزية وأهم عملياتها.
2. تمثل السلاسل الرمزية مستخدماً المصفوفات وتنفذ عملياتها.
3. تمثل السلاسل الرمزية بأسلوب الطول الثابت.
4. تمثل السلاسل الرمزية مستخدماً القوائم المتصلة وتنفذ عملياتها.

3.1 أقسام الوحدة

تتكون هذه الوحدة من ثلاثة أقسام رئيسة ترتبط بقائمة الأهداف السابقة. حيث سنناقش في القسم الأول السلاسل الرمزية باعتبارها تركيبة بيانات تجريدية، إذ سنعرفها ونعرف العمليات الضرورية للتعامل معها، ويرتبط هذا القسم بالهدف الأول. وسنناقش في القسم الثاني كيفية تمثيل السلاسل الرمزية باستخدام المصفوفات ذات البعد الواحد والطول الثابت، ويرتبط هذا القسم بالهدفين الثاني والثالث. أما القسم الثالث فيعرض كيفية تمثيل السلاسل الرمزية باستخدام القوائم المتصلة، وتنفيذ عملياتها، ويرتبط هذا القسم بالهدف الرابع.

4.1 القراءات المساعدة

حاول، عزيزي الدارس، الانتفاع بالمراجع التالية لارتباطها بموضوع هذه الوحدة:

1. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice- Hall. 2000.
2. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison Wesley, 1999.
3. Tremplay, J.P.; and Sorenson, P. G.; An Introduction to Data Structures with Applications, 2nd Edition, McGraw-Hill, 1984

2. السلاسل الرمزية باعتبارها تراكيب تجريدية
Strings As Abstract Data Types

السلسلة الرمزية (String) هي سلسلة من الرموز (Characters) مرتبة بطريقة خطية حيث أن لكل عنصر ترتيباً معيناً، إذ أن هناك عنصراً أولاً، وعنصراً ثانياً، وهكذا. وعدد الرموز المشكلة للسلسلة الرمزية يسمى طول السلسلة. وهنالك عادة قيمة ثابتة تحدد أطول سلسلة رمزية ممكنة MaxString وهي تعتمد على طبيعة البرنامج التطبيقي المستخدم للسلاسل الرمزية.

أمثلة على السلاسل الرمزية:

"Ali Ahmad"

"AbGt678&*()#"

وسنعرف المجال StringRange على أنه جميع القيم المحتملة لترتيب الرمز في السلسلة الرمزية (الأرقام الصحيحة من 0 وحتى MaxString). ويعرف المجال على أنه جميع القيم الممكنة لطول السلسلة الرمزية (أقل طول لسلسلة رمزية هو 0، وأكبر طول لها هو بالطبع MaxString) . ونعرف ذلك في لغة سي++ كما يأتي:

```
const MaxString=80;
const StringRange=MaxString;
```

والعمليات المرتبطة بالسلاسل الرمزية هي:

1. إنشاء السلسلة الرمزية StringCreate

وتستخدم الدالة String Create قبل استخدام السلسلة لأول مرة بهدف تهيئة السلسلة الرمزية للاستخدام بطريقة صحيحة. وطبيعة الخطوات التي تقوم بها تعتمد إلى حد كبير على طريقة تمثيل السلسلة الرمزية.

2. قراءة سلسلة الرمز Read String

وهي دالة تقوم بقراءة السلسلة الرمزية S من جهاز الإدخال، وللدالة الترويسة الآتية:

```
void ReadString(MyString& List);
```

3. كتابة السلسلة الرمزية WriteString

وهي دالة تقوم بكتابة السلسلة الرمزية S على جهاز الإخراج، وللدالة الترويسة الآتية:

```
void WriteString(MyString S);
```

4. إسناد السلاسل الرمزية String Assign

وهي عملية تسند (أو تعين) سلسلة رمزية S إلى متغير T، وهو متغير سلسلي رمزي، بحيث تصبح تلك السلسلة قيمة لذلك المتغير وللدالة الترويسة الآتية:

```
void StringAssign(MyString S,MyString& T);
```

على سبيل المثال لإعطاء المتغير السلسلي الرمزي من نوع (MyString) Name القيمة "Ahmad" فإننا نستدعي الدالة أعلاه كما يأتي:

```
StringAssign("Ahmad",Name);
```

5. إيجاد طول السلسلة الرمزية StringLength

وهي عملية تحسب عدد الرموز في السلسلة S وهي دالة لها الترويسة الآتية:

```
int Length(MyString S);
```

6. وصل السلاسل الرمزية Concatenate

وهي عملية تستخدم لوصل سلسلتين بحيث تصبحا سلسلة واحدة تتكون من رموز السلسلة الأولى S مضافاً إليها رموز السلسلة الرمزية T، وللدالة الترويسة الآتية:

```
void concatenate(MyString& S,MyString T);
```

على سبيل المثال إذا كانت قيمة First هي "Ali" وقيمة Second هي "Ahmad" فإن نتيجة استدعاء concatenate تكون بالطريقة الآتية:

```
concatenate(First,Second);
```

هي جعل

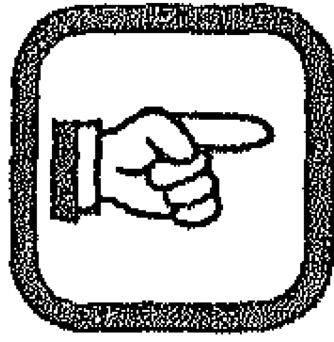
First= "Ali Ahmad"

وتبقى قيمة Second كما هي دون تغيير.

7. استخراج قطع من سلسلة رمزية Substring

وهي عملية تستخدم لاستخراج نسخة عن مقطع في سلسلة رمزية، ولعمل ذلك تحتاج إلى تحديد موقع أول حرف في المقطع START وعدد أحرف المقطع Len.
وللدالة الترويسة الآتية:

```
void Substring(MyString& S,myMaxStr Start,
  myStrRng Len,MyString& T);
```



مثال (1)

إذا كانت قيمة Name هي السلسلة "Ali Ahmad" يمكننا استدعاء Substring كما يأتي:

Substring (Name,5,5, Second)

سيجعل قيمة Second هي المقطع "Ahmad".

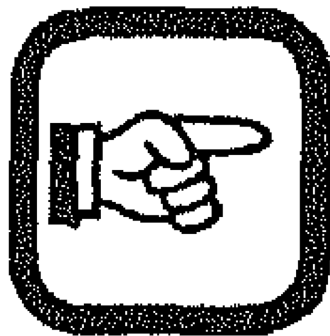
8. البحث عن سلسلة رمزية في سلسلة أخرى StringSearch

وهي عملية تحتاج إليها لمعرفة فيما إذا كانت سلسلة رمزية معينة Sub تظهر في سلسلة أخرى Master كسلسلة جزئية. وتحدد هذه العملية مكان وجود هذه السلسلة Sub في السلسلة الكلية Master وتعيد لنا رقم هذا المكان، وإذا لم تظهر Sub في Master فإن القيمة المعادة هي 0 للإشارة إلى عدم وجود Sub في Master.

ولإعطاء هذه العملية مرونة أكبر فإنها تعطي المستخدم إمكانية تحديد الموقع Start الذي ستتم عملية البحث عن Sub في Master ابتداءً منه.

وعليه فإن للعملية الترويسة الآتية:

```
int StringSearch(MyString Master,MyString sub,
                 myStrRng Start);
```



مثال (2)

إذا كانت

Master = "University of Jordan"
Sub = "of Jordan"

فإن استدعاء العملية StringSearch يتم كما يأتي:

N = StringSearch (Master, Sub, 1)

سيجعل N = 12.

9. إدخال سلسلة رمزية في سلسلة أخرى StringInsert

وتستخدم هذه العملية عند الحاجة إلى إدخال سلسلة رمزية T في أخرى S، بحيث تصبح سلسلة جزئية فيها، وعلى المستخدم أن يحدد مكان الإدخال Place.

وترويسة العملية هي كما يأتي:

```
void StringInsert(MyString& S,MyString& T,myStrRng Place);
```

10. حذف مقطع من سلسلة رمزية StringDelete

وتستخدم لحذف مقطع من سلسلة رمزية وعلى المستخدم أن يحدد عند استدعاء هذه العملية الموقع START الذي يبتدئ عنده المقطع وعدد أحرف ذلك المقطع Number.

void StringDelete(MyString& S,myStrRng Start,
myStrRng Number);

على سبيل المثال إذا كانت قيمة Name هي "Ali Ahmad" واستدعينا العملية StringDelete (Name,5,5) فإنها ستجعل قيمة Name هي "Ali".

11. فحص فيما إذا كانت سلسلتان رمزيتان متساويتين StringEqual

وتعيد هذه العملية القيمة true إذا كانت السلسلتان S وT متساويتين وتعيد false إذا لم تكونا متساويتين. وتنفذ العملية على شكل دالة ذات الترويسة الآتية:

bool StringEqual(MyString S,MyString T);

12. فحص فيما إذا كانت سلسلة رمزية تسبق أخرى حسب الترتيب الأبجدي. LessThan

bool LessThan(MyString S,MyString T);

وتعيد لنا هذه العملية true إذا كانت السلسلة تسبق السلسلة T حسب الترتيب الأبجدي للحروف وحسب قيم ASCII لجميع الرموز.

13. فحص فيما إذا كانت سلسلة رمزية تأتي بعد أخرى GreaterThan

وتعيد لنا هذه العملية true فيما إذا كانت السلسلة S تلي السلسلة T حسب الترتيب الأبجدي.

bool GreaterThan(MyString S,MyString T);

3. تمثيل السلاسل الرمزية بطول ثابت
Fixed Length Implementation of the String

لا شك أنك تعلم، عزيزي الدارس، أن السلاسل الرمزية عادة ما تمثل كمصفوفات مرصوصة (Packed) ذات بعد واحد من الرموز، وطول المصفوفة، كما تعلم، ثابت يحدد عند حجز مواقع المصفوفة ومن هنا كان اسم هذه الطريقة.

وسنستخدم هنا أيضاً المصفوفات الرمزية لتمثيل السلاسل الرمزية ولكننا سنحتفظ بالمصفوفة وعدد رموز السلسلة في صنف class سنطلق عليه اسم MyString وذلك حتى يسهل علينا استرجاع الطول دون الحاجة إلى تنفيذ خطوات كثيرة، وذلك لأن معرفة طول السلسلة الرمزية ضروري للقيام بالكثير من العمليات كعملية الوصل concatenate وغيرها.

حيث الثابت MaxString يمثل أكبر طول لسلسلة رمزية سنتعامل معها.

تنفيذ عمليات السلاسل الرمزية باستخدام أسلوب الطول الثابت:

سنناقش هنا، كيفية تنفيذ بعض عمليات السلاسل الرمزية باستخدام أسلوب الطول الثابت وسنترك تنفيذ بقية العمليات كتدريب لك.

1. إنشاء السلسلة الرمزية StringCreate

كل ما تعمله هذه الدالة هو جعل طول السلسلة صفراً

```
void MyString::StringCreate(MyString& S)
  { S.Len= 0; }
```

2. قراءة السلسلة الرمزية ReadString

```
void MyString::ReadString(MyString& S)
    { int i= MaxString;
    i = 0;
    while (i<=MaxString)
           {
      cin>>S.data[i];
      i++;}
      S.length= i;
    }
```

تقوم هذه الدالة بقراءة السلسلة من سطر ويتوقف الدوران إذا ازداد عدد الرموز المدخلة عن أكبر عدد مقبول MaxString. وداخل الدوران تقرأ السلسلة رمزاً رمزاً وتخزن

في المصفوفة الرمزية Data وتزداد قيمة العداد الذي يعد عدد الرموز المقروءة، وعند انتهاء الدوران يخزن عدد الرموز المقروءة i في الحقل length.

3. كتابة السلسلة الرمزية WriteString

```
void MyString:: WriteString(MyString S)
    {
for (int i= 0; i<S.length;i++)
 cout<<S.data [i];
 }
```

4. إسناد السلاسل الرمزية StringAssign

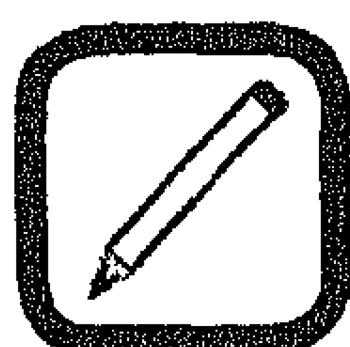
```
void MyString::StringAssign(MyString S,MyString& T)
     { int i=StringRange;
      for(i=0;i<MaxString;i++)
       T.data[i]=S.data[i];
        T.length = S.length;


       }
```

وهي دالة بسيطة تقوم بنسخ المصفوفة S.data فيT.data عنصراً عنصراً. ثم تجعل طول السلسلة T مساوياً لطول السلسلة S.

5. إيجاد طول السلسلة الرمزية Length

وسنترك لك، عزيزي الدارس، تنفيذ هذه العملية البسيطة كتدريب لك.



تدريب (1)

اكتب الدالة Length التي تحسب طول السلسلة الرمزية S. استخدم ترويسة الدالة كما وردت في القسم الأول من الوحدة.

6. وصل السلاسل الرمزية Concatenate

```
void MyString::concatenate(MyString& S,MyString T)
    {int i=StringRange;

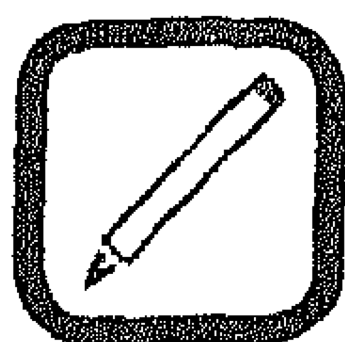
    for(i=0;i<T.length;i++)
     S.data[i+S.length]=T.data[i];

      S.length=S.length + T.length;
   }
```

وتقوم بنسخ السلسلة T في S ابتداءً من آخر موقع في S. ثم تقوم بتعديل طول السلسلة S بحيث يصبح مساوياً للطول القديم زائد طول السلسلة T.

7. استخراج مقطع من سلسلة رمزية SubString

سنترك لك تنفيذ هذه العملية كتدريب لك، عزيزي الدارس.



تدريب (2)

اكتب الدالة SubString بحيث تلتزم بترويسة الدالة كما وردت في القسم الأول من الوحدة.

8. البحث عن سلسلة رمزية في سلسلة أخرى StringSearch

```
int MyString::StringSearch(MyString Master,
            MyString sub,myStrRng Start)
 {int i,j;
  bool found;
  i=Start;
  found=false;
  while (i<Master.length && !found)
   {j=0;
    while (Master.data[i+j-1]==sub.data[j] && !found)
    if (j==sub.length) found:true;
      else j++;
   if (!found)i++;
   }
  if (found) return i;
   else      return 0;
   }
```

تقوم هذه الدالة بمقارنة عناصر السلسلة sub بعناصر السلسلة Master ابتداءً بالموقع Start في Master. فإن توافقت عناصرهما جميعاً يتوقف الدوران، ويعيد الإجراء موقعSub فيMaster . وإن لم يتوافقا تعاد الكرة ولكن ابتداءً من الموقع Start + 1 وهكذا، حتى يتوافقا أو نتأكد من أن subلا يمكن أن تكون سلسلة جزئية في Master.

9. إدخال سلسلة رمزية في سلسلة أخرى StringInsert

```
void MyString::StringInsert(MyString& S,MyString& T,myStrRng Place)
 {int i;
  if (Place<S.length && S.length+T.length <=MaxString)
   { for(i=S.length + T.length;i>=Place;i--)
      S.data[i]=S.data[i-T.length];
     for(i=1;i>=T.length;i++)
      S.data[Place + i-1]=T.data[i];
     S.length=S.length + T.length;
   }
 }
```

تقوم هذه الدالة بالتأكد أولاً من أن عملية الإدخال عملية صحيحة بمعنى أن مكان الإدخال (أقل من طول S) وأن ناتج عملية الإدخال سيبقى سلسلة رمزية ذات طول مناسب أي يمكن أن تخزن في المصفوفة الرمزية ذات الطول MaxString. إن لم يتحقق أي من هذين الشرطين فإن عملية الإدخال غير صحيحة ولا يتم تغيير أي شيء.

أما إن تحقق الشرطان فإن هنالك ثلاث خطوات رئيسة يجب أن تنفذ:

أولاً: يجب عمل فراغ في السلسلة S لوضع السلسلة T وذلك بإزاحة الرموز الموجودة في الموقع Place في S إلى اليمين بمقدار طول السلسلة T وهذا يتم في دوران for الأول.

ثانياً: يتم نسخ عناصر السلسلة T رمزاً رمزاً إلى السلسلة S . وهذا ما يتم من خلال دوران for الثاني.

ثالثاً: يجب أن يتغير طول السلسلة S بحيث تصبح مساوية لطول السلسلة بعد الإضافة وهو الطول القديم زائد طول السلسلة T.

10. حذف مقطع من سلسلة رمزية StringDelete



تدريب (3)

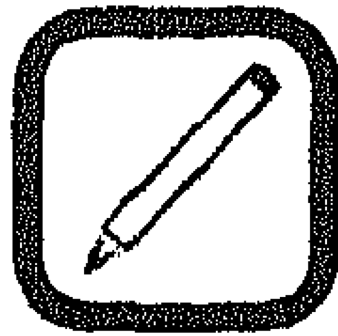
اكتب الدالة StringDelete التي تحذف جزءاً من سلسلة رمزية. يجب أن تتقيد، عزيزي الدارس، بترويسة الدالة كما وردت في القسم الأول من الوحدة.

11. فحص فيما إذا كانت سلسلتان رمزيتان متساويتين StringEqual.

```
bool MyString::StringEqual(MyString S,MyString T)
    {return S.data == T.data;
    }
```

تعلم، عزيزي الدارس، أنك تستطيع مقارنة مصفوفتين رمزيتين في لغة باسكال مباشرة باستخدام إحدى العلاقات البولينية (مثل= >، < = الخ). وعليه فإن كل ما تفعله StringEqual هو مقارنة المصفوفتين الرمزيتين S.data وT.data وإعادة الدالة StringEqual النتيجة كقيمة منطقية.

12. فحص فيما إذا كانت سلسلة رمزية تسبق أخرى حسب الترتيب الأبجدي LessThan.



تدريب (4)

اكتب الدالة LessThan متقيداً بترويسة الدالة الواردة في القسم الأول من الوحدة.

13. فحص فيما إذا كانت سلسلة رمزية تأتي بعد سلسلة أخرى GreaterThan.



تدريب (5)

اكتب الدالة GreaterThan متقيداً بترويسة الدالة الواردة في القسم الأول من الوحدة.

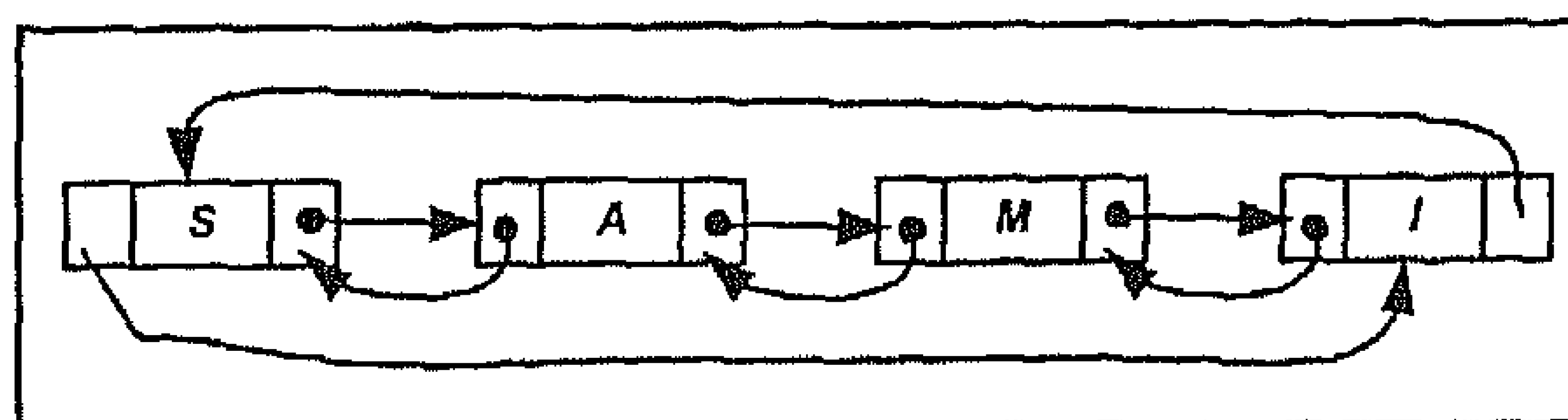
4. تمثيل السلاسل الرمزية باستخدام القوائم المتصلة Linked List Implementation of Strings

في القسم السابق، عزيزي الدارس، ناقشنا استخدام المصفوفات الرمزية ذات الطول الثابت في تمثيل السلاسل الرمزية ولتلك الطريقة مميزات، منها سهولة استخدامها في تمثيل السلاسل وتنفيذ العمليات. ولكن لتلك الطريقة نقاط ضعف أهمها الحاجة إلى تحريك (Shift) أجزاء كبيرة من السلاسل الرمزية عند إدخال مقطع أو شطب مقطع رمزي من تلك السلاسل (انظر كيفية عمل String Insert و String Delete في القسم السابق) مما قد يؤثر كثيراً على وقت التنفيذ اللازم لهذه العمليات.

نقطة ضعف أخرى تكمن في استخدام مصفوفات ذات أطوال ثابتة أن كثيراً من هذه المصفوفات لن تستغل بشكل كامل إذا كانت السلسلة الرمزية قصيرة مما يعني أن هنالك جزءاً من الذاكرة قد يبقى محجوزاً ولكن غير مستغل.

وفي هذا القسم سنناقش، عزيزي الدارس، أسلوباً آخر لتمثيل السلاسل الرمزية قد يكون أعقد قليلاً من الأسلوب السابق، إلا أن الأسلوب الجديد لن يضطرنا إلى تحريك أجزاء من السلاسل عند الإضافة والحذف، وسيستغل الذاكرة بشكل أفضل في كثير من الأحيان.

يستخدم الأسلوب الجديد القوائم المتصلة الثنائية الدائرية لتمثيل السلاسل الرمزية إذ يخزن كل رمز في عقدة (node) وكل عقدة تشير إلى العقدة التالية والعقدة السابقة كما يوضح الشكل (1)، ولكون القائمة دائرية فإن العقدة الأخيرة تشير إلى العقدة الأولى؛ كما تشير العقدة الأولى إلى العقدة الأخيرة، وذلك لتسهيل عملية الإضافة والحذف.



الشكل (1): تمثيل السلسلة "SAMI" باستخدام القوائم الثنائية الدائرية المتصلة

وكما تعلم، عزيزي الدارس، من المهم للتعامل مع القوائم المتصلة الاحتفاظ بعنوان أول عقدة في القائمة، وسنحتفظ أيضاً بعدد العقد في القائمة والذي يمثل أيضاً طول السلسلة الرمزية. سنحتفظ بهذه المعلومات في سجل struct نعطيه الاسم StrNode حسب التعريف الآتي:

```
typedef const int myMaxStr;
typedef const int  myStrRng;
char ch;
myMaxStr MaxString=6;
myStrRng StringRange=MaxString;
class StringPtr
 {private:
    struct StrNode
     { StrNode* next;
       char data;
       StrNode* previous;
     };
    StrNode* head;
    int len;
   public:
    ..............
};
```

سنستخدم head للاحتفاظ بعنوان أول عقدة في القائمة، وlen للاحتفاظ بعدد العقد (طول السلسلة الرمزية) في القائمة. لاحظ، عزيزي الدارس، أن head من النوع StringPtr وهو مؤشر معرّف على أنه مؤشر إلى عقد.

وسنستخدم العقدة من النوع StringNode لتخزين رمز واحد من السلسلة (في الحقل data)، بالإضافة إلى عنوان العقدة التالية (في الحقل next) والتي تحمل الرمز التالي وعنوان العقدة السابقة (في الحقل previous) والتي تحمل الرمز السابق.

تنفيذ بعض عمليات السلاسل الرمزية باستخدام القوائم المتصلة

سنناقش هنا، عزيزي الدارس، كيفية تنفيذ بعض عمليات السلاسل الرمزية بأسلوب القوائم المتصلة، وسنترك لك كتابة بعض هذه العمليات كتدريب لك.

1. إنشاء قائمة StringCreate

```
void StringPtr::StringCreate(StringPtr& S)
  {S.head=NULL;
   S.len=0;
   }
```

كل ما تقوم به هذه الدالة هو إنشاء سلسلة فارغة، أي أن طول السلسلة صفر، ولعدم وجود أي عقدة فيها فإن قيمة head هي NULL.

2. قراءة سلسلة رمزية StringRead

```
void StringPtr::StringRead(StringPtr& S)
 { StrNode *p,*q;
   int i;
   cin>>ch;
   i=1;
   p=new StrNode;
   S.head = p;
   p->previous =p;
   p->next =p;
   p->data =ch;
   q=p;
   i=1;
   while (i<MaxString)
    {cin>>ch;
     p = new StrNode;
     p->data=ch;
     q->next=p;
     q=p;
     i++;
    } //make the doubly linked list circular
   p->next=S.head;
   S.head->previous=p;
   S.len=i;
 }
```

3. كتابة سلسلة رمزية StringWrite

```
void StringPtr::StringWrite(StringPtr S)
 {StrNode * p;
  p=S.head;
  int i=0;
  if (p!=NULL)
    do
     {cout<<p->data;
      p=p->next;
      i++;
      }
    while(i<=S.len);
    cout<<endl;
 }
```

4. إسناد السلاسل الرمزية StringAssign



تدريب (6)

اكتب الدالة StringAssign وتقيد بترويسة الدالة الواردة في القسم الأول من الوحدة.

5. إيجاد طول السلسلة الرمزية LengthString

وهي دالة بسيطة تعيد قيمة الحقل len

```
int StringPtr::LengthString(StringPtr S)
  {
    return S.len;
  }
```

6. وصل السلاسل الرمزية Concatenate



تدريب (7)

اكتب الدالة concatenate وتقيد بترويسة الدالة الواردة في القسم الأول من الوحدة.

7. استخراج مقطع من سلسلة رمزية Substring .



تدريب (8)

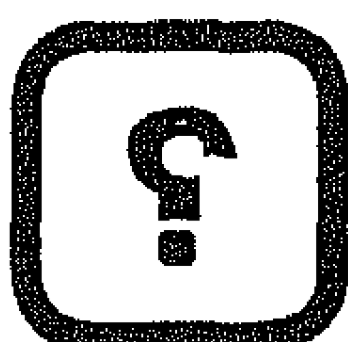
اكتب هذه الدالة وتقيد بترويسة الدالة كما وردت في القسم الأول من الوحدة.

8. إدخال سلسلة رمزية في سلسلة أخرى StringInsert

```
void StringPtr::StringInsert(StringPtr& S,
                    StringPtr T,int place)
 { StrNode *p,*temp;
  int        i;
  StrNode *BeginningOfT, *LastOfT;
  if (place<=S.len)
```

```
  {p=S.head;
   for(i=1; i<place;i++)
     p=p->next;
   BeginningOfT=T.head;
   LastOfT=T.head->previous;
   BeginningOfT->previous=p->previous;
   LastOfT->next=p;
   p->previous->next=BeginningOfT;
   p->previous=LastOfT;
   S.len=S.len+T.len;
  }//end if
}
```

يقوم هذا الإجراء أولاً بالتأكد من أن عملية الإضافة صحيحة بمعنى أن مكان الإضافة place مكان مقبول ويكون مقبولاً إذا كان أقل أو يساوي طول السلسلة التي سنضيف إليها سلسلة أخرى. ثم يستخدم دوران for لوضع المؤشر p على مكان الإضافة، أي العقدة (node)، التي ستضاف عندها السلسلة الجديدة. وبعد ذلك يبقى وصل السلسلتين بحيث تصبحان سلسلة واحدة.



أسئلة التقويم الذاتي (1)

1. اذكر مجموعة العمليات التي تجرى على السلاسل الرمزية.
2. وضح الفرق بين تمثيل السلاسل الرمزية الممثلة بأسلوب الطول الثابت والممثلة بأسلوب القوائم المتصلة.
3. علل السبب في أن السلاسل الرمزية الممثلة بأسلوب الطول الثابت لا تستغل الذاكرة بطريقة مثلى.
4. وضح لماذا نحتاج إلى الاحتفاظ بعدد العقد في السلسلة الرمزية الممثلة بأسلوب القوائم المتصلة.
5. أجب بنعم أو لا على كل من العبارات التالية:

أ. السلسلة الرمزية ترتب بطريقة خطية.

ب. طول السلسلة الرمزية يساوي عدد الرموز المشكلة للسلسلة الرمزية.

ج. لاستخراج مقطع من سلسلة رمزية فقط نحتاج تحديد أول رمز في المقطع.

د. دالة فحص تساوي سلسلتان رمزيتان تعيد قيمة منطقية.

5. الخلاصة

ناقشنا، عزيزي الدارس، في هذه الوحدة السلاسل الرمزية (Strings) وأهم العمليات الضرورية لاستخدامها، وناقشنا أيضاً تمثيل هذه السلاسل بأسلوبين هما: أسلوب الطول الثابت باستخدام المصفوفات الرمزية، وأسلوب القوائم المتصلة.

وتوصلنا إلى أن أسلوب الطول الثابت (باستخدام المصفوفات) بالرغم من سهولة برمجته، يعاني من بعض السلبيات مثل الحاجة إلى نقل بعض الرموز عند الإضافة أو الحذف باتجاه اليمين أو اليسار. بالإضافة إلى أن هذا الأسلوب قد لا يستغل الذاكرة بطريقة مثلى إذ قد تحجز مصفوفة، ولا تستخدم جميع مواقعها. بالمقارنة فإن أسلوب القوائم المتصلة لا يحتاج إلى نقل الرموز إلى مواقع جديدة عند عملية الحذف أو الإضافة، مما يقلل من الوقت اللازم لإجراء هاتين العمليتين. وبالنسبة للذاكرة، فإن أسلوب القوائم المتصلة يحجز عقدة لكل رمز عند الحاجة؛ لذلك لن تجد عقدة محجوزة دون سبب. ولكن لاحظ، عزيزي الدارس، أن كل حرف يحتاج إلى عقدة وكل عقدة تحتوي على ثلاثة حقول: Previous وData وNext مما يعني الحاجة إلى كم كبير من الذاكرة لكل حرف. وللتقليل من شأن هذه المشكلة، نستطيع تخزين أكثر من رمز في كل عقدة.

6. لمحة عن الوحدة الدراسية السادسة

في الوحدة القادمة، عزيزي الدارس، سنناقش موضوع الطوابير (Queues) باعتبارها تراكيب بيانات تجريدية، وأهم تطبيقاتها العملية.

7. إجابات التدريبات

تدريب (1)

```
void MyString::Length(MyString S)
{
 length=S.length;
}
```

تدريب (2)

```
void MyString::subString(MyString& S, int len,
                          int start,   MyString& T,)
   {int i;

   for(i=1;i<=T.len;i++)
    T.data[i]=S.data[start+1-1];

     T.length=len;
  }
```

تدريب (3)

```
void MyString::StringDelete(MyString& S,myStrRng Start,
                             myStrRng Number)
 {int i;
    If (Start<=S.length && (Start+Num<=S.length+1))
     {
       for(i=Start;i<=MaxString-Number;i++)
      S.data[i]=S.data[i+Number];

                S.length=S.length-Number;
   }
```

تدريب (4)

```
bool MyString::LessThan(MyString S,MyString T)
    {
       return (S.dat<T.data) ;
     }
```

تدريب (5)

```
bool MyString:: GreaterThan (MyString S,MyString T)
  {
    return (S.dat>T.data) ;
   }
```

تدريب (6)

```
void  StringPtr::StringAssign(StringPtr S,StringPtr& t)
  { StrNode *Sptr,*Tptr, *prev;
   Sptr=S.head;
   StringCreate(t);
   if (S.head!=NULL)
    { Tptr=new StrNode;
```

```
  t.head=Tptr;
  Tptr->data=Sptr->data;
  prev=Tptr;
  while(Sptr->next=S.head)
    {cout<<"in loop";
     Sptr=Sptr->next;
     Tptr=new StrNode;
     Tptr->data=Sptr->data;
     prev->next=Tptr;
     Tptr->previous=prev;
     prev=Tptr;
    }
  Tptr->next=t.head;
  t.head->previous=Tptr;
  t.len=S.len;
 }

}
```

تدريب (7)

```
void StringPtr::concatenate(StringPtr& S,StringPtr T)
 { StrNode *LastInT,*LastInS;
  if (S.len+T.len<MaxString)
   {//let LastInT point to last node of string T
    LastInT=T.head->previous;

    //let LastInS point to last node of string S
    LastInS=S.head->previous;

    //link the Last node of S point to first node of string T
    LastInS->next=T.head;

   //link the first node in S to the last node in string T
    S.head->previous=LastInT;

    //link the Last node in T  to the first node in string S
    LastInT->next=S.head;

    //link the first node in T to the last node in string S
    T.head->previous=LastInS;

    //adjust the length of S
```

```
 S.len=S.len+T.len;
 }
}
```

تدريب (8)

```
void StringPtr::Substring(StringPtr S,int Start,int len,StringPtr& T)
{ StrNode *Sptr,*Tptr,*prev;
  int i;
  if (Start+len<S.len)
  { Sptr=S.head;
    //position Sptr at the first node to be copied
    for(i=1; i<Start;i++)
       Sptr=Sptr->next;
    //copy the first character and make it head of T
    Tptr=new StrNode;
    Tptr->data=Sptr->data;
    T.head=Tptr;
    prev=Tptr;
    //copy the last of the characters
    for(i=Start+1;i<Start+len-1;i++)
     {Sptr=Sptr->next;
      Tptr=new StrNode;
      Tptr->data=Sptr->data;
      Tptr->previous=prev;
      prev->next=Tptr;
      prev=Tptr;
     }
    while(Sptr->next=S.head)
     {cout<<"in loop";
      Sptr=Sptr->next;
      Tptr=new StrNode;
      Tptr->data=Sptr->data;
      prev->next=Tptr;
      Tptr->previous=prev;
      prev=Tptr;
     }
    //make T a circular linked list
    Tptr->next=T.head;
    T.head->previous=Tptr;

  }
}
```

8. مسرد المصطلحات

- سلاسل رمزية Strings: السلسلة الرمزية هي سلسلة من الرموز (Characters) مرتبة بطريقة خطية حيث أن لكل عنصر ترتيباً معيناً، إذ أن هناك عنصراً أولاً، وعنصراً ثانياً، وهكذا.

- قوائم متصلة Linked Lists: نوع من أنواع تمثيل القوائم وفق علاقة ما لتخزين معلومات في كل عنصر عن موضع العنصر التالي في سياق محدد.

- عقد Nodes: العقدة هي بمثابة معلومات تخص عنصر ما من عناصر القائمة المتصلة.



9. المراجع

1. Tremplay, J.P.; and Sorenson, P. G.; An Introduction to Data Structures with Applications, 2nd Edition, McGraw-Hill, 1984.
2. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice- Hall. 2000.
3. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison- Wesley, 1999.



الطوابير Queues

محتويات الوحدة

1 . المقدمة

1.1 تمهيد

أهلاً بك، عزيزي الدارس، إلى الوحدة السادسة من كتاب "تركيب البيانات وتصميم الخوارزميات"، وهي بعنوان "الطوابير". تعالج هذه الوحدة أحد الموضوعات المهمة وهو الطوابير. وقد تمت مناقشة هذا الموضوع في نطاق ثلاثة محاور رئيسة: الأول هو تحديد مفهوم الطوابير وأسلوب تمثيلها في ذاكرة الحاسوب، والثاني كيفية إجراء عملية الإضافة والحذف والثالث هو الاستخدامات والتطبيقات المختلفة للطوابير.

وقد تم تزويد الوحدة بالعديد من الأمثلة والرسومات التوضيحية التي تهدف إلى وضع الأفكار التي نوقشت ضمن التصورات المنطقية والفيزيائية الصحيحة لها. كما تم تزويد الأقسام المختلفة بالعديد من التدريبات وأسئلة التقويم الذاتي، التي تهدف إلى ترسيخ المفاهيم المختلفة في ذهن الدارس. وبالإضافة إلى هذا وذاك فإن هناك عدداً من الخوارزميات والبرامج الفرعية المصاحبة التي تهدف إلى بيان التفاصيل البرمجية الدقيقة لإجراء العمليات المختلفة على الطوابير وللتعامل مع بعض الاستخدامات العملية. وقد تم تقديم خوارزميات الإضافة والحذف بأساليب مختلفة تبعاً لأسلوب التمثيل في الذاكرة وللمنهج المتبع في التعامل مع الطوابير.

أهلاً بك، عزيزي الدارس، مرة أخرى إلى رحاب هذه الوحدة. ونرجو أن تستمتع بدراستها وتنتفع بما تضمنته من مسائل وأفكار. ونحن بانتظار مشاركتك واستفساراتك حولها.

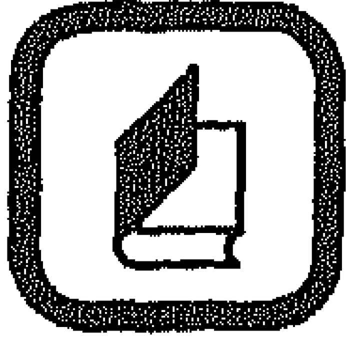
2.1 أهداف الوحدة

ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1. توضح المفاهيم الأساسية المرتبطة بالطوابير.
2. تبين طرق تمثيل الطوابير واستخداماتها.
3. تكتب الخوارزميات الخاصة بعمليات الطوابير.
4. تشرح مفهوم الطوابير الدائرية، وطوابير الأولوية والطوابير المزدوجة وتكتب الخوارزميات المختلفة التي تتعلق بها.

3.1 أقسام الوحدة

تتضمن هذه الوحدة أربعة أقسام رئيسة ترتبط بقائمة الأهداف السابقة. فالقسم الأوّل يوضح المفهوم المجرد للطوابير، ويرتبط هذا القسم بالهدف الأوّل من أهداف الوحدة. أما القسم الثاني فقد عرض هذه المفاهيم وما يتصل بها من مسائل برمجية حول الطوابير بشكل مفصل وبأسلوب عملي، وتشمل هذه المسائل طرق تمثيل الطوابير واستخداماتها. ويرتبط هذا القسم بالهدف الثاني من أهداف الوحدة. وأما القسم الثالث فقد عني بالخوارزميات الخاصة بالعمليات المستخدمة على الطوابير وهذا يحقق الهدف الثالث. وسندرس في القسم الرابع أنواعاً مختلفة من الطوابير مثل الطوابير الدائرية، وطوابير الأولوية والطوابير المزدوجة. وهذا يغطي الهدف الرابع.



4.1 القراءات المساعدة

على الرغم من شمولية الأقسام والمسائل التي تم عرضها في هذه الوحدة إلا أن هناك فوائد كثيرة يمكن أن يجنيها الدارس من الرجوع إلى بعض القراءات الإضافية المساعدة. فهناك المزيد من المعلومات والأمثلة في هذه المصادر، وهناك أيضاً المزيد من التدريبات والأسئلة. ونوصي بالمصدرين التاليين لهذه الغاية، مع التذكير بأن قائمة المراجع في نهاية هذه الوحدة تتضمن مصادر على قدر كبير من الأهمية والفائدة:

1. Amsbury, Wayne, Data structures from Arrays to Priority Queues. Belmont (USA): Wadwsorth, 1985. pp. 97-119, 120-141, & 169-198.
2. Malik, D.S. Data Structures Using C++, 1st Edition, Course Technology, Inc., 2003.
3. Main, Michael Data Structures & Other Objects Using C++, 3rd Edition, Addison-Wesley, 2004.

2. مفاهيم ومصطلحات أساسية

لقد لاحظنا، عزيزي الدارس، في حديثنا السابق عن القوائم المتصلة بأن عمليات إضافة القيم الجديدة إلى هذه القوائم وحذف القيم منها يمكن أن تتم في أي مكان في القائمة سواء أكان أولها أو وسطها أو نهايتها. ومعنى ذلك أنها على النحو الذي وصفت به في الوحدة السابقة، قد لا تناسب بعض المشكلات التي تلتزم نهجاً محدداً في الإضافة أو الحذف كالطوابير مثلاً.

فالطابور تركيب خطي يسير على مبدأ إضافة العناصر الجديدة إلى نهاية القائمة وحذف العناصر من بدايتها. وهذا المبدأ يتطابق مع ما نمارسه في الحياة العملية العامة. ففكرة الطوابير مطبقة في الكثير من الممارسات اليومية. ومن أمثلة ذلك صفوف انتظار حافلات الركاب، وصفوف الانتظار في البنوك، وصفوف انتظار المركبات والمسافرين في نقاط العبور والخروج. وكل هذه الأمثلة وغيرها مما يشبهها تلتزم بالقاعدة العامة القائلة: من يأتي أولاً يُخدم أولاً! ولذلك يشار للطابور أحيانا باسم الفيفو (FIFO) وذلك اختصاراً للتعبير الإنجليزي (First In First Out).

ومن المهم أن نشير في هذا المقام إلى مفهوم له علاقة مباشرة بما نحن بصدده الآن، أعني نظرية الطوابير (Queueing theory). وموضوع اهتمام هذه النظرية هو دراسة السلوك المتعلق بحركة الطوابير: كيف تنشأ؟ وما حجم نموها؟ وما مدى إمكانية المورد أو الموارد المتاحة للاستجابة لهذا النمو؟ وما هو الحد الأعلى المقرر لحجم الطابور في ضوء هذه الإمكانية؟ هذه المسائل وغيرها مما له صلة بنظرية الطوابير تدخل ضمن مجال الدراسات الإحصائية، ولها تطبيقاتها المباشرة في نظم التشغيل التي تدير عمل أجهزة الحاسوب وفي شبكات المعلومات والاتصالات.

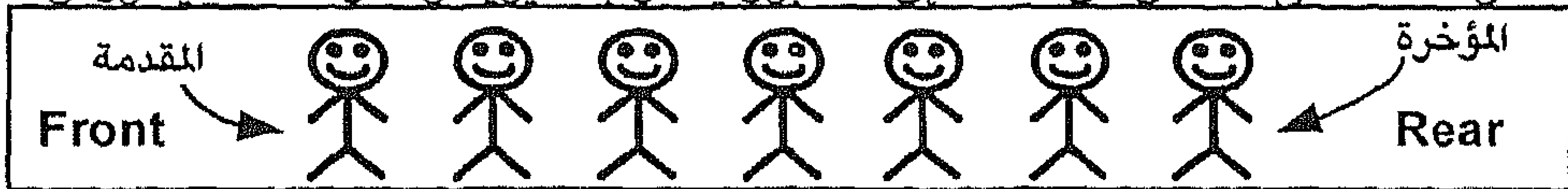
لقد جاءت فكرة التركيب البياني، الذي نشير إليه باسم الطابور (أو صف الانتظار) من هذه النظرية. ولكن ينبغي أن نلاحظ بأن هذا التركيب البياني وما يبنى عليه من عمليات وخوارزميات لا يهتم بالمسائل والتعقيدات التي تنطوي عليها حركة البيانات بداخله على النحو الذي نجده في نظرية الطوابير. وجل اهتمامنا هنا ينصب على كيفية تخزين البيانات في هذا التركيب البياني وكيفية الوصول إليها.

وكما أن الطابور (بالمعنى العام الذي نعرفه) له بداية وله نهاية فكذا الحال بالنسبة للطابور، بالمعنى الاصطلاحي الذي نحن بصدده. وبينما نشير إلى بداية الطابور بمصطلح

المقدمة (FRONT) فإننا نشير إلى نهاية الطابور، وهو آخر عنصر فيه بمصطلح المؤخرة (REAR). ومن هنا، فإن الإضافة تتم من ناحية المؤخرة، بينما يتم الحذف أو الإلغاء من جهة المقدمة. فلو تأملنا طابوراً من الأشخاص ينتظرون الدخول إلى إحدى طائرات نقل المسافرين، وذلك على النحو الموضح في الشكل (1)، لوجدنا أن أول شخص في الطابور (أي مقدمة الطابور) هو أول شخص يصعد إلى الطائرة، وأن آخر شخص في هذا الطابور (أي المؤخرة) هو آخر من يصعد إليها.

الشكل (1): مقدمة ومؤخرة طابور من المسافرين

وينبغي أن لا يفهم من كلامنا هذا بأن الطابور ثابت في الطول وفي طبيعة القيم المكونة له. فعلى العكس من ذلك، فإن الطابور يمتاز بالحيوية والحركة. فقد يطول وقد



ولعلك تلاحظ مما ذكر أعلاه بأن الطوابير، بصفتها تراكيب بيانية، ما هي إلا شكل من أشكال القوائم تلتزم بمبدأ محدد في عمليات الإضافة أو الحذف.

أسئلة التقويم الذاتي (1)



أجب بنعم أو لا:

1. الطابور تركيب بياني يسمح بالإضافة والحذف في كلا الاتجاهين: البداية والنهاية.
2. يطلق على بداية الطابور اسم المقدمة (FRONT) بينما يطلق على نهايته اسم المؤخرة (REAR).
3. ينصب اهتمام التراكيب البيانية التي يطلق عليها اسم الطوابير على المسائل التي تتصل بنظرية الطوابير.
4. يمكن تمثيل الطوابير باستخدام أحد الأسلوبين المعروفين للتخزين: المتصل والتتابعي.

3. الطوابير واستخداماتها وطريقة تمثيلها

لقد ذكرنا فيما سلف بأن الطابور، بالمعنى الذي نحن بصدده، تركيب بياني يظهر على شكل قائمة من القيم. ومعنى ذلك أن التصور المنطقي لا يختلف، من وجهة نظر مبدئية، عن التصور الذي عرضناه عند حديثنا عن القوائم. ولتوضيح هذا التصور، دعنا، عزيزي الدارس، نتأمل الشكل (2).

FRONT REAR

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}

الشكل (2): تصور منطقي للطابور

ففي هذا الشكل ثمّة مجموعة من العناصر الممثلة بالقيمة (Xi). ويشار إلى العنصر الأول فيها بمؤشر خاص يعبر عن مقدمة الطابور وهو (FRONT) ويستخدم لأغراض الحذف. كما يشار إلى العنصر الأخير بمؤشر خاص آخر يعبر عن مؤخرة الطابور وهو (REAR) ويستخدم لأغراض الإضافة.

وفي ضوء هذا المفهوم الخاص والتصور المنطقي للطابور فإن هناك بعض السمات الأساسية التي يتسم بها الطابور وتجعله قادراً على محاكاة الواقع الفعلي الذي يحاول ترجمته والتعبير عنه، وأهم هذه الخصائص ما يلي:

أ. يتم تزويد الطابور، عند البدء ببنائه، بالوسيلة المناسبة للإشارة إلى مقدمته وإلى مؤخرته لتسهيل عمليات الإضافة والحذف.

ب. عندما يتساوى عدد مرات الحذف مع عدد القيم المضافة إلى الطابور، فإن الطابور يصل إلى وضع يصبح معه خالياً من القيم.

ج. أن القيمة الوحيدة التي يسمح بالوصول إليها ومعالجتها في الطابور هي القيمة الأولى ولكي يتم الوصول إلى العناصر الأخرى ومعالجتها، فلا بد من استرجاع قيمة العنصر الأول ومعالجته قبل كل شيء.

د. أن حجم الطابور متغير وذلك اعتماداً على ما يضاف إليه وما يحذف منه، ومن هنا فإننا نصفه بالديناميكية.

هـ. يمكن وصف عملية الوصول إلى العناصر في الطابور بأنها مباشرة. وبذلك فإنها لا تستغرق وقتاً كبيراً. بيد أن هذا الوصول محدود بمقدمة الطابور ومؤخرته. وهذا بالطبع على خلاف الاسترجاع والمعالجة، فهما يتحددان بالبداية وليس بالنهاية.

وتبرز أهمية الطوابير بشكل خاص في عمل نظم التشغيل المستخدمة في أجهزة الحاسوب التي تعمل وفق مبدأ اقتسام الوقت (time-sharing systems) وتقدم خدماتها لأكثر من مستفيد. إذ لا بد في هذا الوضع من تنظيم عملية استخدام الموارد المتوفرة والتي تتمثل بوحدة الحساب والمنطق والذاكرة وأجهزة الطباعة والذاكرة المساعدة وأي أجهزة أخرى مصاحبة. وبذلك فإن من الممكن أن يكون هناك عدد من الطوابير بعدد الموارد المطلوبة.

فهناك، على سبيل المثال، طابور خاص بالطباعة. والذي يحدث عادة هو أن النتائج الخاصة بأحد البرامج توضع في وحدة الذاكرة المساعدة ريثما تتوفر الإمكانية لطباعتها. ومعنى ذلك أنه سيتكون لدينا طابور بكل البرامج التي تنتظر نتائجها في الداخل ريثما تحين الفرصة لطباعتها. فإذا افترضنا أن لدينا جهازاً واحداً للطباعة وأنه ليس هناك أي أفضليات لطباعة عمل قبل آخر، فإن أول عمل ينتهي هو أول عمل يطبع. واعتماداً على سرعة جهاز الطباعة وحجم المعلومات المراد طباعتها، فإن الطابور يزداد ويتقلص. فهناك أوقات قد يكون فيها الطابور خالياً من الأعمال، وهناك أوقات أخرى قد يصل فيها حجم الطابور إلى حد تضطر معه وحدة المعالجة المركزية إلى إيقاف عملها مؤقتاً ريثما يصبح بالإمكان إنجاز بعض أعمال الطباعة المنتظرة وتقليص حجم الطابور، وبالتالي إفساح المجال للأعمال الجديدة.

وهناك أكثر من أسلوب يمكن أن يتبعه نظام التشغيل لتنظيم عمل الطابور. وأحد هذه الأساليب هو حجز جزءٍ مناسب من الذاكرة لتخزين الطابور نفسه. ويتضمن هذا الطابور معلومات أساسية مثل: رقم العمل، ومكان تخزينه في الذاكرة المساعدة، وكمية المعلومات التي يتضمنها العمل الواحد (أي عدد الأسطر التي ستطبع). ويبدأ نظام التشغيل بوضع الأعمال تباعاً في الطابور. وكلما فرغ جهاز الطباعة من عمل معين، فإنه ينتقل إلى العمل التالي له في الطابور. ولكي يتجنب نظام التشغيل المشكلات الناجمة عن الحاجة إلى إزاحة العناصر الموجودة في الطابور باتجاه بداية المساحة المخصصة له في الذاكرة من أجل إفساح المجال لإضافة العناصر الجديدة إلى الطابور فإنه يمكن أن يتعامل مع المواضع المحجوزة في الذاكرة على أنها تشكل طابوراً دائرياً. وسنرى فيما بعد كيف يمكن تطبيق فكرة الطوابير الدائرية.

وأياً كان الأسلوب المتبع في تنظيم عمل الطابور، فإن من المشاكل الأساسية التي تبرز هي تقدير حجم الطابور. فكما ذكرنا أعلاه، قد يصل الطابور إلى وضع الفائض

(overflow) الأمر الذي يؤدي إلى توقف مؤقت لعمل وحدة المعالجة المركزية. ولتجنب مثل هذا الوضع أو التقليل من حالات وقوعه فإن من الممكن القيام بتقدير الحجم المتوقع للطابور وبالتالي حجز مساحة في الذاكرة تتناسب مع درجة نمو الطابور. ويدخل في هذا التقدير ثلاثة عوامل رئيسة هي: معدل الإضافة، ومعدل الحذف، والزمن اللازم لإجراء المعالجة أو تقديم الخدمة.

ولكن ينبغي أن نذكر بأن مشكلة الفائض (أو امتلاء الطابور) والحاجة إلى تقدير الحجم المناسب للطابور تبرز بصفة خاصة في التمثيل القائم على استخدام المصفوفات أو ما أطلقنا عليه قبل قليل اسم التمثيل التتابعي. فالطابور يمكن أن يمثل في الذاكرة باستخدام المؤشرات (القوائم المتصلة) أيضاً. ونظراً لأن هذا النوع من التمثيل يقوم على فكرة الحجز الديناميكي للذاكرة فإن مشكلة الوصول إلى وضع الفائض غير واردة. وفيما يلي نستعرض كيفية تمثيل الطوابير باستخدام المصفوفات أولاً ومن ثم باستخدام المؤشرات:

1.3 العمليات المستخدمة على الطوابير

هنالك العديد، عزيزي الدارس، من العمليات التي تستخدم الطوابير من خلالها، وهذه العمليات هي:

1. إنشاء طابور QueueCreate

وتستخدم هذه الدالة لتهيئة طابور للاستخدام لأول مرة، وهي دالة لها الترويسة التالية:

void QueueCreate(MyQueue& Q);

وطبيعة الخطوات التي تقوم بها هذه الدالة تعتمد على طريقة تمثيل الطابور (باستخدام المصفوفات أو القوائم المتصلة).

2. فحص فيما إذا كان الطابور ممتلئاً QueueFull

وهي دالة بولينية (Boolean) ترجع القيمة البولينية true إذا كان الطابور ممتلئاً، ولا يتسع للمزيد من العناصر وترجع false فيما عدا ذلك. وهي دالة لها الترويسة التالية:

bool QueueFull(MyQueue Q);

3. فحص فيما إذا كان الطابور فارغاً QueueEmpty

وهي أيضا دالة بولينية (Boolean) ترجع القيمة البولينية true إذا كان الطابور فارغاً، والقيمة false إذا لم يكن فارغاً.

وللعملية الترويسة التالية:

bool QueueEmpty(MyQueue Q);

4. إضافة عنصر إلى الطابور EnQueue

وهي دالة تقوم بإضافة عنصر إلى الطابور شريطة أن لا يكون الطابور ممتلئاً. وللعملية الترويسة التالية:

void EnQueue(MyQueue& Q, int element);

لاحظ أن نوعية العنصر المضاف تعتمد على طبيعة التطبيق الذي يستخدم الطابور.

5. حذف عنصر من الطابور DeQueue

هي دالة تقوم بحذف عنصر من الطابور شريطة أن لا يكون الطابور فارغاً وللعملية الترويسة التالية:

void DeQueue(MyQueue& Q, int& element);

لاحظ، عزيزي الدارس، إن المعامل element يستخدم لإعادة العنصر الذي حذف من بداية الطابور إلى البرنامج المستدعي للعملية.

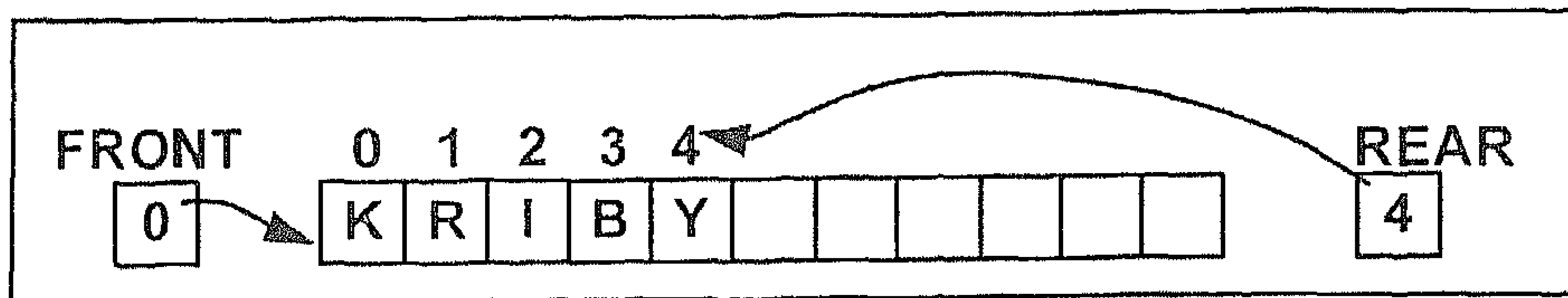
2.3 تمثيل الطوابير

هنالك طريقتان رئيستان لتمثيل الطوابير هما التمثيل التتابعي والتمثيل المتصل.

تستخدم طريقة التمثيل التتابعي المصفوفات أحادية البعد أما طريقة التمثيل المتصل فتستخدم القوائم المتصلة.

1.2.3 تمثيل الطوابير باستخدام المصفوفات (Queues as Arrays)

هذا النوع من التمثيل هو ما نشير إليه باسم التمثيل التتابعي. وهو يقوم على استخدام المصفوفات الأحادية البعد لتخزين الطابور وذلك على النحو الموضح في الشكل (3). وكما تلاحظ في هذا الشكل، فإن فكرة هذا النوع من التمثيل يقوم على وجود مصفوفة أحادية بعدد (MaxSize) من المواقــع ووجــود متغيــرين يشير أحدهما إلى بداية الطابور والآخر يشــير إلى نهاية الطابور.



الشكل (3): تمثيل الطابور باستخدام المصفوفات

ولعلك تلاحظ أيضاً بأن قيمة المتغير (REAR) لا تساوي MaxSize في هذه الحالة. كما أن قيمة المتغير (FRONT) لا تساوي واحد (1) أو الدالة الدنيا للمصفوفة في كل الحالات. فبينما تتحدد قيمة الأول في ضوء عمليات الإضافة، فإن قيمة المتغير الثاني تتحدد في ضوء عمليات الحذف.

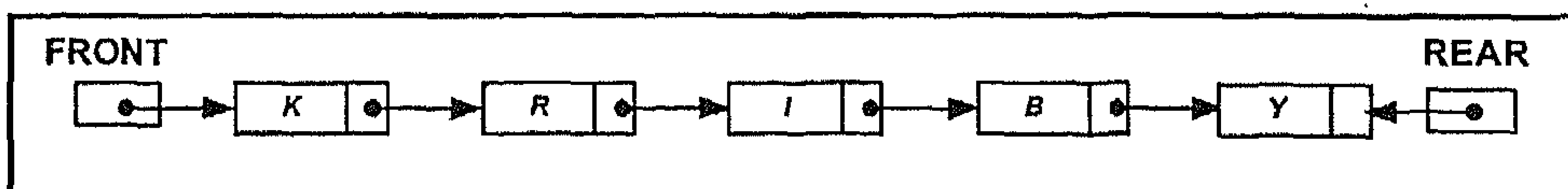
ويمكن التعبير عن هذا النوع من التمثيل بالصيغة البرمجية التالية:

```
typedef const int myMaxQue;
myMaxQue MaxSize=12;
 class MyQueue {
  private:
    int data [ MaxSize];
    int FRONT,REAR;
  public:
     ...
};
```

2.2.3 تمثيل الطوابير باستخدام القوائم المتصلة

(Queues as Linked lists)

يقوم هذا النوع من التمثيل على استخدام المؤشرات التي توفرها لغات البرمجة (مثل لغة ++C)، ويسمح للمبرمج بتخزين الطابور على شكل قائمة متصلة مفردة وذلك على النحو الموضح في الشكل (4). وعلى خلاف أسلوب التمثيل التتابعي، فإن هذا النوع من التمثيل لا يحجز في الذاكرة إلا عدداً من المواضع مساوياً لعدد القيم التي يشتمل عليها الطابور في لحظة معينة. وهو بذلك انعكاس طبيعي لحقيقة الطابور ونهجه العملي.

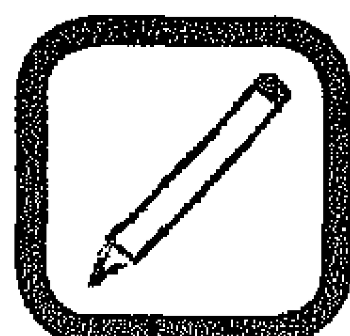


الشكل (4): تمثيل الطابور باستخدام القوائم المتصلة

أما من الناحية البرمجية فلا يكاد يختلف عن القوائم المتصلة المفردة. فكل عنصر من عناصر الطابور يتكون من سجل يتضمن حقلين: أحدهما للقيمة البيانية والأخرى للمؤشر. وبالإضافة إلى ذلك لدينا مؤشران خارجيان، هما: المقدمة (FRONT) والمؤخرة (REAR). ويمكن التعبير عن هذه الصيغة البرمجية على النحو التالي:

```
class QPtr
    {public:
     int value;
     QPtr* next;
   };
class MyQueue
 {public:
     int size;
     QPtr* FRONT;
     QPtr* REAR;
...
};
```

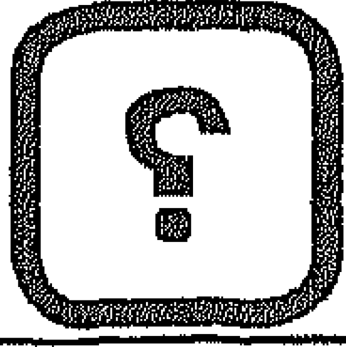
وكما فعلنا في التمثيل التتابعي، حددنا النمط البياني لعناصر الطابور (value) أنه من النوع int لكنه بإمكانك تحديد نوع النمط البياني الذي يلزمك، فقد يكون ذلك char أو int أو float أو غير ذلك من الأنماط البيانية. وقد يتم التعبير عن المعلومات المتعلقة بالعنصر الواحد بأكثر من حقل.



تدريب (1)

افترض أن إحدى عيادات الأطباء تحتفظ بقائمة لأسماء المراجعين وفقاً لتسلسل مجيئهم إلى العيادة في كل يوم ويقوم الطبيب باستدعائهم بناءً على هذه القائمة، وافترض أيضاً أن العيادة لا تتسع لأكثر من عشرة أفراد في الوقت الواحد، مما يعني إبقاء عدد المسجلين في القائمة بهذه الحدود. والمطلوب منك هو كتابة التعريفات اللازمة بلغة سي++ للتعبير عن هذا الطابور أوّلاً باستخدام التمثيل التتابعي وثانياً باستخدام التمثيل المتصل.

أسئلة التقويم الذاتي (2)



أجب بنعم أو لا:

1. القيمة الوحيدة التي يسمح بالوصول إليها ومعالجتها في الطابور هي قيمة العنصر الأول.
2. يسمح الطابور بالوصول إلى القيم المخزنة تباعاً ابتداء من بداية الطابور أو نهايته.
3. عندما يتساوى عدد مرات الحذف مع عدد مرات الإضافة فإن ذلك يؤدي إلى خلو الطابور من القيم.
4. عدد عناصر الطابور ينبغي أن يتساوى مع عدد المواضع المحجوزة له في الذاكرة في حالة التمثيل باستخدام المصفوفات.
5. عدد عناصر الطابور يتساوى مع عدد المواضع المحجوزة له في حالة التمثيل المتصل باستخدام المؤشرات.
6. إن قيمة المتغير (FRONT) ينبغي أن تكون مساوية للدالة (supscript) الدنيا للمصفوفة.
7. تستخدم فكرة الطوابير في جدولة الأعمال الداخلة إلى جهاز الحاسوب والخارجة منه، في حالة تعدد المستفيدين (multi-user system).
8. التمثيل المتصل باستخدام المؤشرات يعكس الطبيعة الحقيقية والنهج العملي للطابور بشكل أفضل من التمثيل التتابعي باستخدام المصفوفات.

4. تنفيذ العمليات المستخدمة على الطوابير

في هذا القسم سنناقش، عزيزي الدارس، كيفية تنفيذ عمليات الطوابير آخذين بعين الاعتبار الطريقتين المختلفتين لتمثيل الطوابير.

1.4 انشاء الطابور QueueCreate

كما سبق وأشرنا تقوم هذه الدالة بتهيئة الطابور للاستخدام لأول مرة بجعله طابوراً خالياً.

في حالة التمثيل التتابعي (المصفوفات) كل ما يجب عمله هو جعل FRONT = -1 و REAR = -1 وعليه فإن الإجراء هو:

```
void MyQueue::QueueCreate(MyQueue& Q)
  { Q.FRONT= -1;
    Q.REAR= -1;
  }
```

أما في حالة التمثيل المتصل فإن ما يقوم به الإجراء هو جعل FRONT=NULL ، و REAR=NULL كما يلي:

```
void MyQueue::QueueCreate()
  {FRONT=NULL;
   REAR=NULL;
  }
```

2.4 فحص فيما إذا كان الطابور ممتلئاً QueueFull

في حالة التمثيل المتتابع فإن الطابور يكون ممتلئاً إذا كانت جميع مواقع المصفوفة مستخدمة. ويكون ذلك في حالة أن FRONT=0 و REAR=-1 وعليه فإننا نستطيع كتابة QueueFull كما يلي:

```
bool MyQueue::QueueFull(MyQueue Q)
      { if(Q.FRONT==0 && Q.REAR==-1)
                    return true;
                     else
                    return false;
      }
```

أما في حالة التمثيل المتصل فإن حجم الطابور يعتمد على حجم الذاكرة. ويعتبر الطابور ممتلئاً عندما تفشل العملية new في تعيين موقع جديد، يضمه إلى باقي عناصر الطابور.

3.4 التحقق فيما اذا كان الطابور فارغاً QueueEmpty

يكون الطابور فارغاً في حالة التمثيل التتابعي إذا كانت FRONT=0 و REAR= 0 وعليه فإن الدالة QueueEmpty يمكن كتابتها كما يلي:

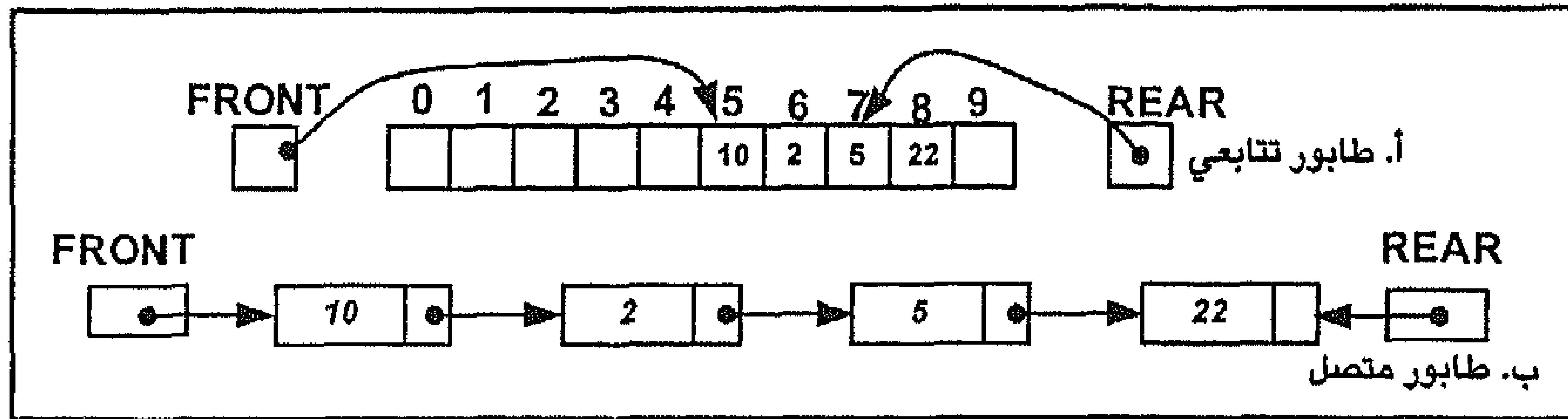
```
bool MyQueue::QueueEmpty(MyQueue Q)
    { if(Q.FRONT== -1 && Q.REAR== -1)
            return true;
        else return false;
    }
```

أما في حالة التمثيل المتصل فيكون الطابور فارغاً إذا كانت FRONT= NULL و REAR= NULL وعليه نستطيع كتابة QueueEmpty كما يلي:

```
bool MyQueue::QueueEmpty()
  {if(FRONT==NULL && REAR==NULL)
     return true;
     else return false;
  }
```

4.4 عملية الإضافة إلى الطوابير (Enqueue)

تتم عملية الإضافة إلى الطابور في اتجاه واحد هو مؤخرة الطابور. وهي تتأثر إلى حد كبير بأسلوب التمثيل المستخدم في الذاكرة، وبمعدل الحذف الذي يتم على الطابور. ولتوضيح التأثير الناتج عن اختلاف أسلوب التمثيل، دعنا، عزيزي الدارس، نتأمل الشكل (5). ففي هذا الشكل لدينا طابور ممثل بأسلوبين مختلفين: الأول تتابعي (باستخدام المصفوفات) والآخر متصل (باستخدام القوائم المتصلة).

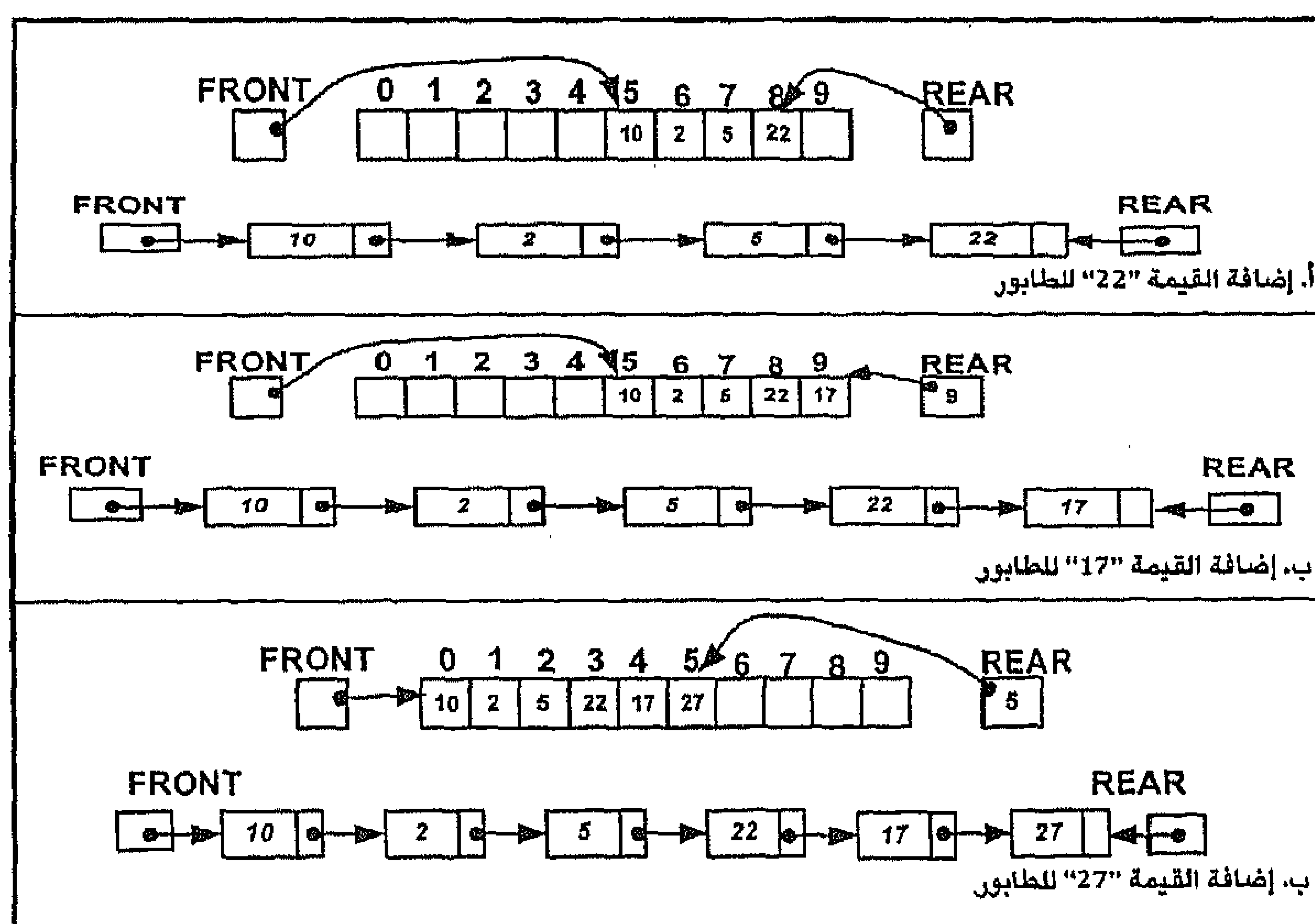


الشكل (5): طابور ممثل بأسلوبين مختلفين

ويتضمن هذا الطابور ثلاث قيم. ولعلك تلاحظ بأن الطابور قد تعرض لعملية الحذف وبذلك وصل إلى الوضع الذي هو عليه. وهذا يتضح بصورة خاصة في تمثيل الطابور باستخدام المصفوفة.

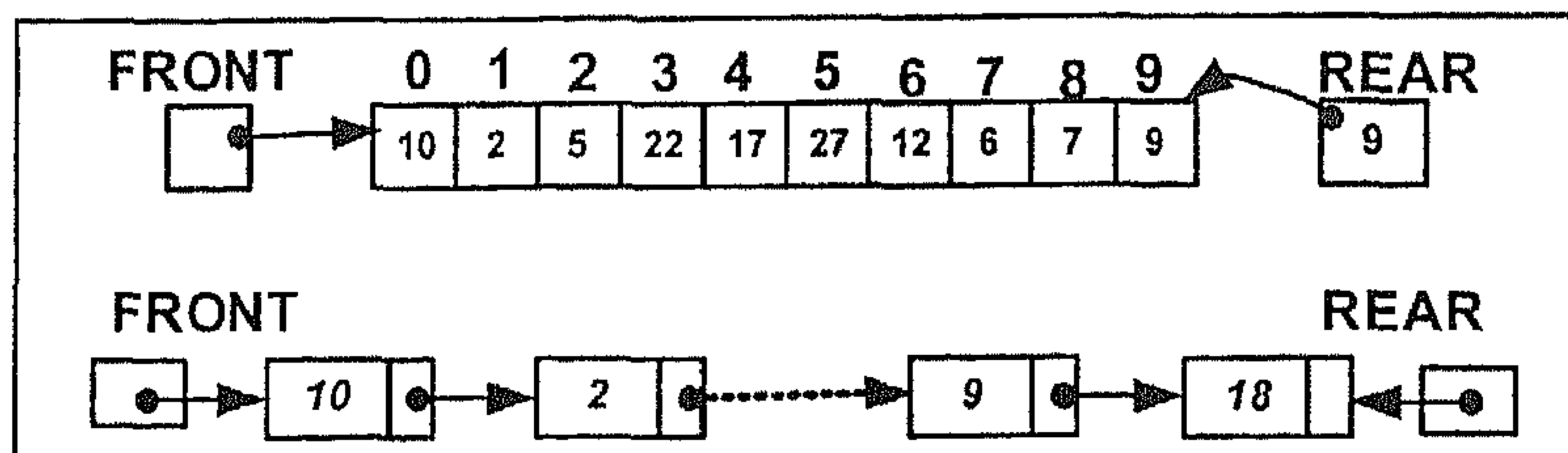
والآن افترض أننا نرغب في إضافة القيم الثلاثة التالية على التوالي (من اليمين إلى اليسار) وهي: 22، 17، 27. ففي حالة إضافة القيمة الأولى سيصبح وضع الطابور على النحو الموضح في الجزء الأول من الشكل (6). وكما تلاحظ من هذا الشكل، فإن هذه القيمة قد أضيفت إلى الموضع التاسع في المصفوفة، كما تمت إضافتها إلى نهاية القائمة المتصلة، وتم تعديل قيمة المتغير (REAR) بناءً على هذه الإضافة، بحيث أصبح في وضع يشير فيه إلى العنصر قبل الأخير في الطابور. وقد فعلنا الشيء نفسه عند إضافة القيمة الثانية، كما هو موضح في الجزء الثاني من الشكل (6)، وبذلك تم تعديل قيمة المتغير (REAR) بحيث أصبح في وضع يشير فيه إلى العنصر الأخير في الطابور.

وقد كان من نتيجة إضافة هاتين القيمتين أن أصبحت قيمة المتغير (REAR) مساوية للحد الأعلى للمصفوفة وهو (1-). ومعنى ذلك أننا وصلنا إلى وضع لا يسمح بإضافة القيمة الثالثة دون القيام بعمل معين، وهو وضع شبيه بالفائض (OVERFLOW). وهذا على خلاف الطابور لدى تمثيله بشكل متصل، حيث الإمكانية متوفرة للإضافة دون الوصول إلى وضع من هذا النوع. وبناء على هذا الوضع الجديد فقد اضطررنا إلى إزاحة جميع القيم الموجودة في الطابور باتجاه بداية المصفوفة، من أجل إتاحة المجال لإضافة القيمة الجديدة "27" وأي قيم جديدة أخرى. وبناءً على ذلك فقد تم تعديل قيمة كل من المتغيرين (REAR) و (FRONT)، فأصبحت قيمة الأول "6" بعد أن كانت "10"، وأصبحت قيمة الثاني "1". في حين بقيت قيمة هذين المتغيرين ثابتة في حالتي الإضافة السابقتين على القيمة "6".



الشكل (6): إضافة القيمة "27" إلى الطابور

والآن، عزيزي الدارس، دعنا نضيف القيم التالية تباعاً وذلك في ضوء الوضع الذي وصل إليه الطابور حتى هذه اللحظة، وهي (12، 6، 7، 9، 18). ودعنا نفترض أيضاً أنه لن يكون هناك حذف لأي قيمة في هذه الأثناء. إن الذي يحدث هو كما يلي: ستتم إضافة القيم الأربعة الأولى دون أي مشكلات. وفي كل مرة يتم تعديل قيمة المؤشر (REAR) مع بقاء قيمة المؤشر (FRONT) ثابتة طيلة ذلك كله. ولكن عندما نريد إضافة القيمة الأخيرة تبرز لدينا مشكلة مهمّة في حالة استخدام المصفوفات هي وصول الطابور إلى وضع الفائض الحقيقي، إذ لم يعد هناك متسع لإضافة أي قيم أخرى. وهذا الوضع لا يحصل عند التمثيل باستخدام المؤشرات، حيث تتم إضافة القيمة الخامسة (18) دون أي مشكلة. ويمكن التعبير عن هذا الوضع بالشكل (7).



الشكل (7): الطابور في وضع الفائض في حالة المصفوفة فقط

وفي ضوء هذه الأوضاع المختلفة التي يمر بها الطابور وبخاصة في حالة استخدام المصفوفات، وفي ضوء التغير الذي يحدث على المؤشر (REAR) فإن عملية الإضافة يمكن التعبير عنها بالخوارزميتين (1) و (2) حيث الخوارزمية (1) تتعلق بالتمثيل التتابعي وتتعلق الخوارزمية (2) بالتمثيل المتصل واستخدام المؤشرات.

الخوارزمية (1):

```
void MyQueue::EnQueue(MyQueue& Q, int element)
    {//inserting a new element into an array-based queue
    int i, last;
      if(Q.FRONT==0 && Q.REAR== -1)
          cout<<"queue is full .. overflow";
    else
      {if (Q.FRONT== -1)  // first element
        {Q.FRONT=0;
         Q.REAR=1;
        } // end IF
        else
         if (Q.REAR== -1) // shift elements to the left)
           {last=1;
```

```
        for (i=Q.FRONT;i<=Q.REAR;i++)
          {data[last]=data[i];
           last=last+1;
          }//end FOR
        Q.REAR= Q.REAR - Q.FRONT + 1;
        Q.FRONT=1;
      }// end IF
          Q.REAR=Q.REAR + 1;
      data[Q.REAR]= element;
      }// end ELSE
}
```

وكما تلاحظ، عزيزي الدارس، فإن هذه الخوارزمية التي تعالج مسألة إضافة أحد العناصر إلى طابور ممثل في الذاكرة على شكل مصفوفة تنطوي على أربعة أوضاع مختلفة، نوجزها فيما يلي:

أولاً: ليس هناك أي متسع لإضافة أي عناصر جديدة، وبذلك يوصف الطابور بأنه قد وصل إلى وضع الفائض. ويستدل على ذلك من خلال قيمة كل من المتغيرين (FRONT) و (REAR) ، حيث تم التعبير عن ذلك بالاختبار الشرطي:

if(Q.FRONT==0 && Q.REAR== -1)

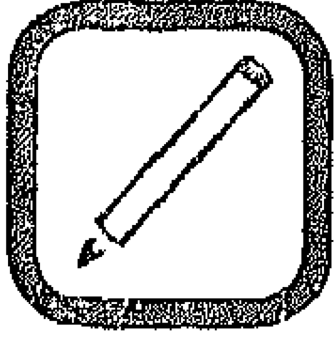
ثانياً: إن الطابور خال من القيم، وبذلك فإن القيمة المضافة تشكل العنصر الأول في الطابور ويستدل على خلوه من القيم من خلال قيمة المتغير (Q.FRONT). وقد تم التعبير عن هذا الوضع في الخوارزمية بالاختبار الشرطي:

if (Q.FRONT== -1)

ثالثاً: لم يعد هناك متسع لإضافة أي قيم جديدة إلى نهاية المصفوفة إلا أن هناك أماكن شاغرة في بدايتها. وبذلك فإنه لا بد من إعادة تنظيم عملية التخزين من خلال إزاحة القيم الموجودة إلى بداية المصفوفة، وبالتالي تحويل الأماكن الشاغرة إلى نهاية المصفوفة بدلاً من أولها. ويستدل على هذا الوضع من خلال قيمة المتغير (REAR)، حيث يتم التعبير عن ذلك في الخوارزمية بالاختبار الشرطي التالي:

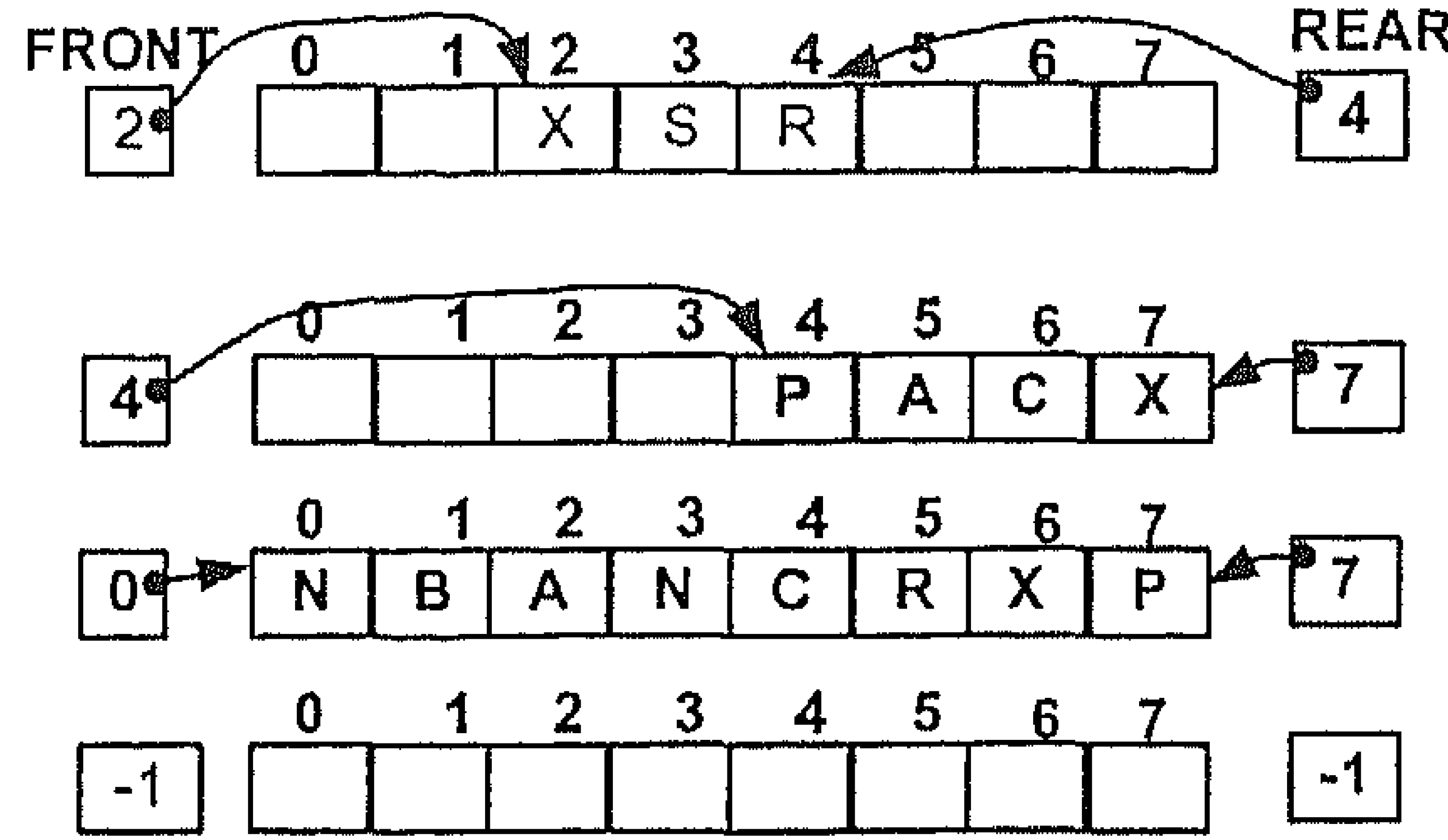
if (Q.REAR== -1)

رابعاً: إن هناك مجالاً لإضافة قيمة جديدة إلى الطابور في مكانها المناسب. وهذا هو الوضع الاعتيادي الذي نصل إليه في الخوارزمية، إذا لم يتحقق أي من الاختبارات الشرطية الثلاثة السابقة.



تدريب (2)

لعلك لاحظت أن خوارزمية الإضافة تنطوي على أربعة أوضاع مختلفة تمّ تحديدها وتفسيرها، والآن أمامك، عزيزي الدارس، أربعة طوابير مختلفة في الشكل (8). فتأمّلها جيداً وبيّن أيّاً منها ينطبق على إحدى الحالات الأربعة للإضافة الواردة أعلاه، ثم قارن إجابتك بالإجابات المعطاة في نهاية الوحدة.

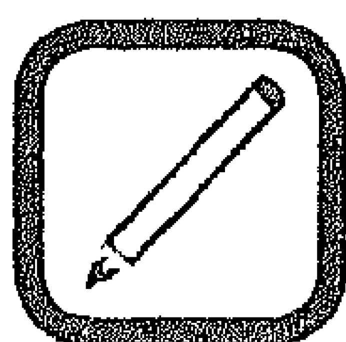


الشكل (8): طابور في أوضاع مختلفة للإضافة

بعد أن أوضحنا كيف يمكن أن نقوم بعملية الإضافة باستخدام التمثيل التتابعي للطوابير، دعنا نرى كيف يمكن أن نقوم بالعملية ذاتها على الطوابير الممثلة في الذاكرة باستخدام المؤشرات والقوائم المتصلة. وهنا ينبغي أن نذكر بأنه ليس لدينا أي من الأوضاع المختلفة التي ذكرناها أعلاه فيما يتصل باستخدام المصفوفات. فإما أن يكون الطابور خالياً (empty)، وبذلك فإن قيمة كل من المؤشرين (REAR) و (FRONT) هي NULL، أو يكون هناك في الطابور عنصر واحد أو أكثر، وبذلك فإن الإضافة تتم بسهولة كبيرة. وفيما يلي نستعرض، عزيزي الدارس، الخطوات التي تنطوي عليها عملية الإضافة من خلال الخوارزمية (2).

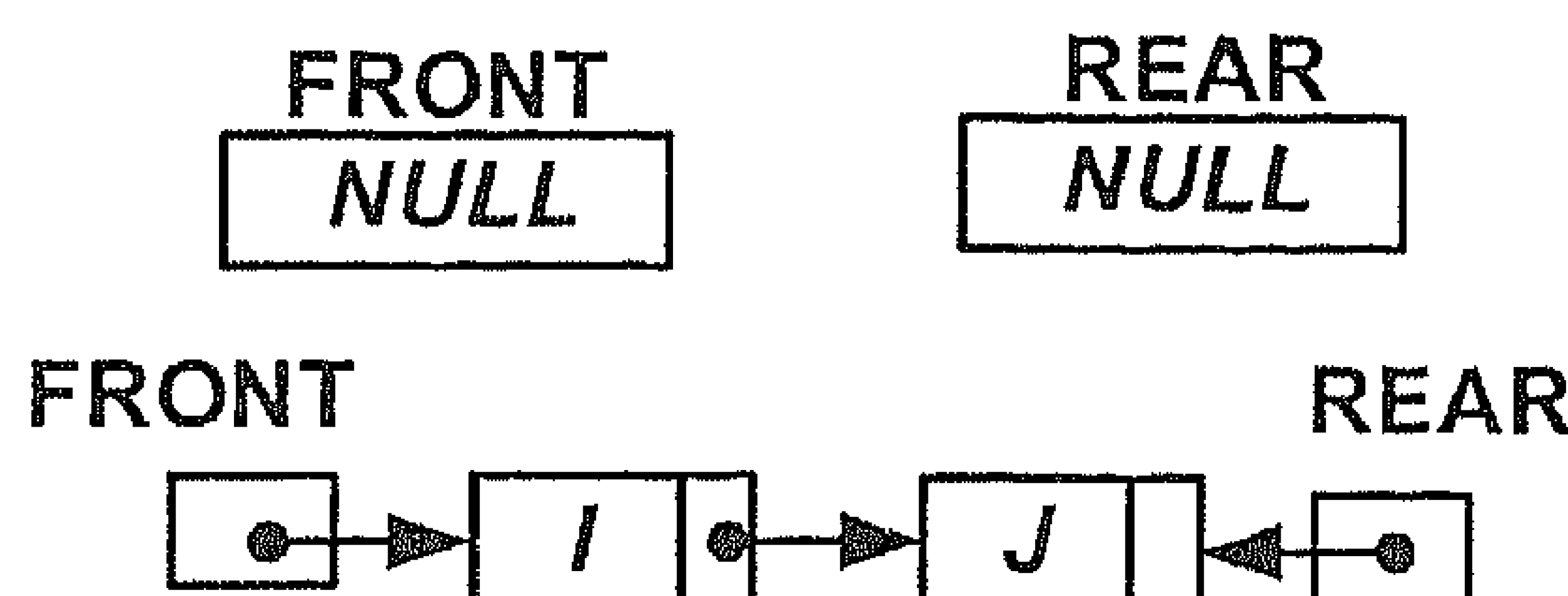
الخوارزمية (2):

```
void MyQueue::EnQueue(int element)
 {//inserting a new element into a queue represented as a linked list
  QPtr  *newNode;
  newNode= new QPtr; // create a new node
  newNode->value=element;
  newNode->next= NULL;
    if (REAR==NULL) // queue is empty
     FRONT= newNode; //insert as first element
     else
       REAR->next=newNode; // insert at end
       REAR= newNode;
   }
```



تدريب (3)

استخدم الطابورين المذكورين أدناه في الشكل (9) لبيان الوضع الذي سينتهي إليه كل من هذين الطابورين بعد إجراء الإضافة في ضوء الخوارزمية (2) المذكورة أعلاه. قدم الإجابة على هذا التدريب باستخدام الرسم أولاً، ثم ببيان أي من شقي التركيب الشرطي في الخوارزمية سينفذ.



الشكل (9): طابور في وضعين مختلفين

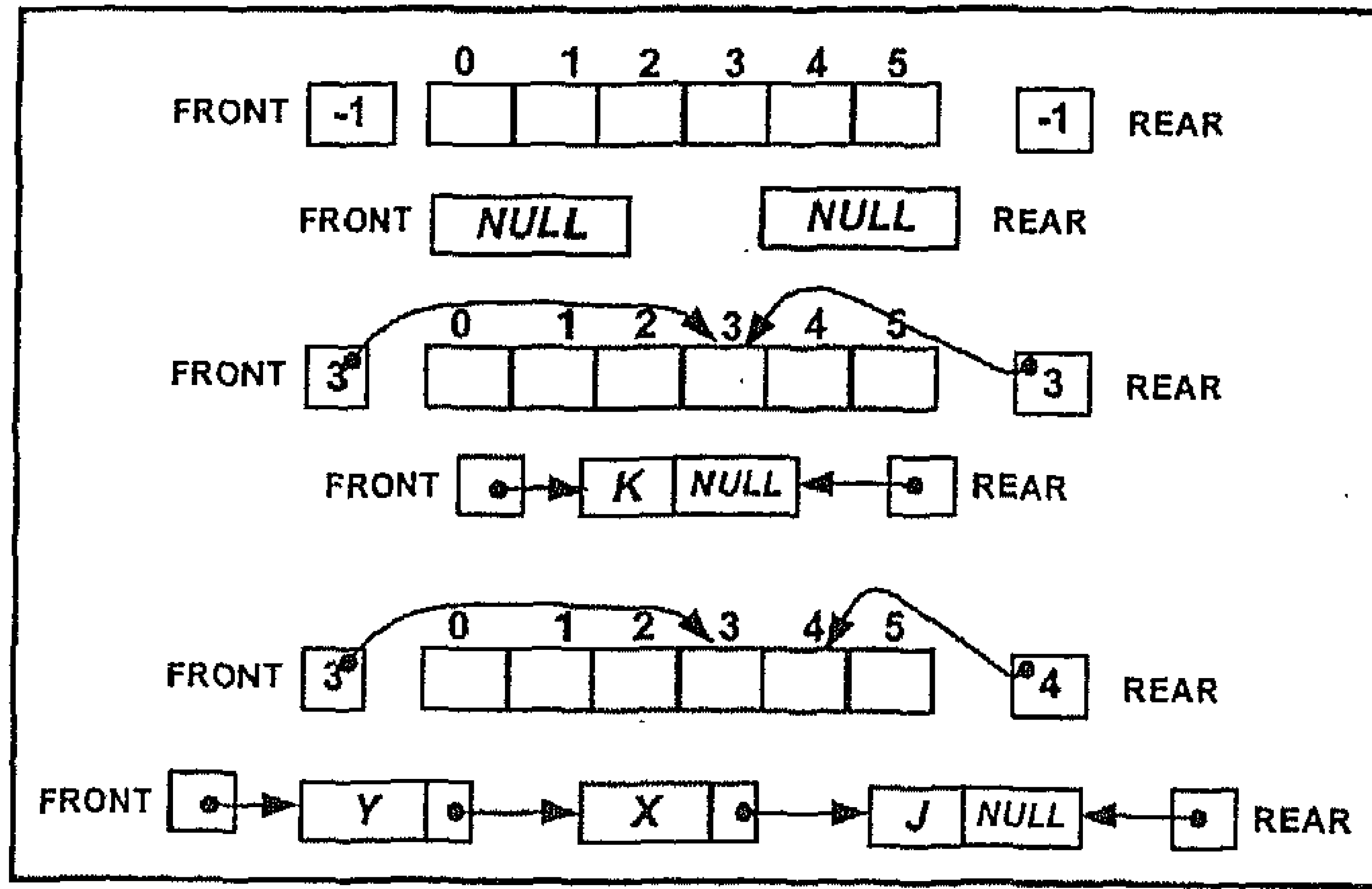
ولو حاولنا تتبع ما يحدث بالنسبة لهاتين الحالتين لوجدنا، أن الجمل الثلاثة الأولى ستنفذ على كل حال. وبذلك يتم حجز المكان المناسب للعنصر الجديد ثم إسناد القيمة إليه. والفرق الوحيد يكمن في شقي التركيب الشرطي. فكما ترى أنّ الحالة الأولى تعبر عن طابور خال من القيم. وبذلك فإن الإجابة على السؤال: (REAR==NULL ?) هي بالإثبات. ومن هنا، يتم تنفيذ جواب الشرط مباشرة. وبذلك يصبح العنصر المضاف هو العنصر الوحيد في الطابور. وخلو الطابور من القيم قد يعني شيئاً واحداً من اثنين: الأول إن الطابور لم يدخل مرحلة الاستعمال والنشاط بعد، وبذلك يكون العنصر المضاف هو العنصر الأول والوحيد. أما الثاني فهو أن الطابور في مرحلة النشاط الفعلي وأن عملية الحذف قد تساوت مع عملية الإضافة وبذلك تم حذف جميع العناصر التي دخلت الطابور حتى الآن.

أما الحالة الثانية، فإنها تمثل الوضع الاعتيادي للطابور. وهو وضع يتسم بالحركة والزيادة والنقصان. وطالما أن قيمة المؤشر (Q.REAR) ليست NULL، فإن الشق الثاني للتركيب الشرطي المعبر عنه بكلمة else هو ما سيتم تنفيذه. وهذا الأمر لا يختلف باختلاف عدد العناصر. فسواء تضمن الطابور عنصراً واحداً أو تضمن عشرات العناصر فإن أسلوب تنفيذ الخوارزمية لن يتغير في هذا الشأن.

5.4 عملية الحذف من الطوابير (Dequeue)

تتم عملية الحذف في الطوابير في اتجاه واحد هو مقدمة الطابور. وتتأثر عملية الحذف بأسلوب تمثيل الطوابير في الذاكرة، شأنها في ذلك شأن الإضافة. كما أنها، من زاوية أخرى، تؤثر على عملية الإضافة، كما أسلفنا من قبل. ذلك أن حذف أحد العناصر يفسح المجال لإضافة عنصر آخر. ولكن ينبغي أن نذكر بأن تأثير أسلوب التمثيل لا يصل إلى المستوى نفسه الذي لمسناه في عملية الإضافة. والاختلاف القائم بين التمثيل التتابعي والتمثيل المتصل لا يتجاوز مسألة الصياغة البرمجية. أما أوضاع الحذف بحد ذاتها فلا تعكس أي اختلاف ملموس بين الأسلوبين.

وبصرف النظر عن أسلوب التمثيل المستخدم، فإن هناك ثلاثة أوضاع عامة لحذف العناصر من الطوابير: ينجم الأول نتيجة لخلو الطابور من القيم، وينجم الثاني نتيجة لوجود عنصر واحد في الطابور، والوضع الثالث هو الوضع الاعتيادي للحذف. ولكي يتضح الفرق بين هذه الأوضاع المختلفة، دعنا نتأمل الشكل (10) الذي يعرض هذه الأوضاع باستخدام أسلوبي المصفوفات والقوائم المتصلة.



الشكل (10): الطابور في الأوضاع المختلفة للحذف

ففي الحالة الأولى الممثلة بالطابور الخالي ليس لدينا، أو لم يعد لدينا، في الطابور أي قيم نتعامل معها. وبذلك تتوقف عملية الحذف ريثما تدخل الطابور قيم أخرى، ويتم تنشيط عملية الحذف من جديد. وكما تلاحظ فإننا نستدل على هذا الوضع من خلال قيمة أي من المتغيرين (FRONT) أو (REAR) اللذين يعبران عن هذا الوضع بالقيمة صفر أو NULL.

أما في الحالة الثانية، فإن لدينا عنصراً واحداً في الطابور. وهذا يتضح من تساوي

قيمة المتغيرين (FRONT) و (REAR)، مع الأخذ بالاعتبار أن هذه القيمة ليست صفراً أو NULL. فكلا المؤشرين يشيران إلى هذا العنصر الوحيد. ومعنى ذلك أن إجراء الحذف على هذا العنصر سيؤدي إلى إخلاء الطابور من القيم والتحول إلى الوضع الذي تم عرضه في الحالة الأولى.

وأخيراً، فإن لدينا الوضع الاعتيادي والمتمثل بوجود قيمتين أو أكثر في الطابور ومعنى ذلك أن حذف إحدى القيم من الطابور لن يؤدي إلى إخلائه وبالتالي فإن التعديل يقتصر على المتغير (FRONT).

وفيما يلي نعرض لتفاصيل التعامل مع هذه الأوضاع الثلاثة من خلال الخوارزميتين التاليتين. وكما فعلنا بالنسبة لعملية الإضافة فإن الخوارزمية (3) تتعلق بأسلوب تمثيل الطوابير باستخدام المصفوفات، والخوارزمية (4) تختص بالطوابير الممثلة على شكل قوائم متصلة.

الخوارزمية (3):

```
void MyQueue::DeQueue(MyQueue& Q, int& element)
    {//deleting an element from an array-based queue
   if(Q.FRONT== -1)
     cout<<"the queue is empty";
     else
       if (Q.FRONT==Q.REAR) //only one element in the queue
            {element=Q.data[Q.FRONT];
        Q.FRONT= -1;
        Q.REAR= -1;
        } // END IF
        else
          {element=Q.data[Q.FRONT];
           Q.FRONT=Q.FRONT + 1;
           } // end ELSE
  }
```

وكما تلاحظ، عزيزي الدارس، فإن الخطوات التي تعرضها هذه الخوارزمية تعكس بالفعل الأوضاع الثلاثة المذكورة أعلاه. ويتضح هذا من طبيعة التراكيب الشرطية المستخدمة والتي اتخذت سلسلة من (if-then-else). ثم لاحظ أيضاً أننا احتفظنا بالقيمة المحذوفة مؤقتاً من أجل أي معالجة مطلوبة عليها.

الخوارزمية (4):

```
void MyQueue::DeQueue(int element)
  {// deleting an element from a queue represented as a linked list
   if (FRONT==NULL)
    cout<<" the queue is empty";
    else
```

```
if (FRONT==REAR) // one element
  {element=FRONT->value;
   FRONT=NULL;
   REAR=NULL;
  }// end IF
else
  {element=FRONT->value;
  FRONT=FRONT->next;
  }//              end ELSE
}
```

وبمقارنة هذه الخوارزمية بالخوارزمية السابقة ندرك مدى التشابه بينهما من حيث معالجة مسألة الحذف. فالفرق بين عملية الحذف القائمة على استخدام المصفوفات وعملية الحذف القائمة على استخدام القوائم المتصلة هو فقط في صيغة الجمل، أما منطق العملية وتسلسل الخطوات فهو متطابق تماماً.

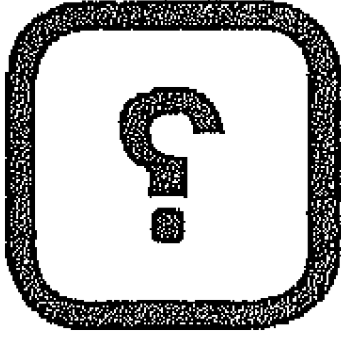


تدريب (4)

استخدم البيانات المعطاة في الجدول التالي وبين بالرسم - من خلال كتابة برنامج وتنفيذه على الحاسوب - وضع الطابور بعد إجراء عملية الحذف أو عملية الإضافة لكل قيمة من القيم المذكورة وفق ما هو محدد في الجدول باستخدام الخوارزميات الأربعة التي تمت مناقشتها أعلاه. وهذا يعني أن عملك سيكون مزدوجاً: مرة باستخدام المصفوفات ومرة أخرى باستخدام القوائم المتصلة، وذلك لكل عنصر من العناصر المذكورة أدناه. وافترض لغايات التمثيل باستخدام المصفوفات إن قيمة MaxSize هي "6".

لاحظ أن العمليات تتم على التوالي وفق التسلسل المعطى في الجدول.

العمليــة	القيمة	
اضافة	R	0
اضافة	E	1
اضافة	M	2
اضافة	X	3
حذف	R	4
حذف	E	5
اضافة	B	6
اضافة	Y	7
اضافة	L	8
اضافة	Z	9



أسئلة التقويم الذاتي (3)

إملأ الفراغ في العبارات التالية:

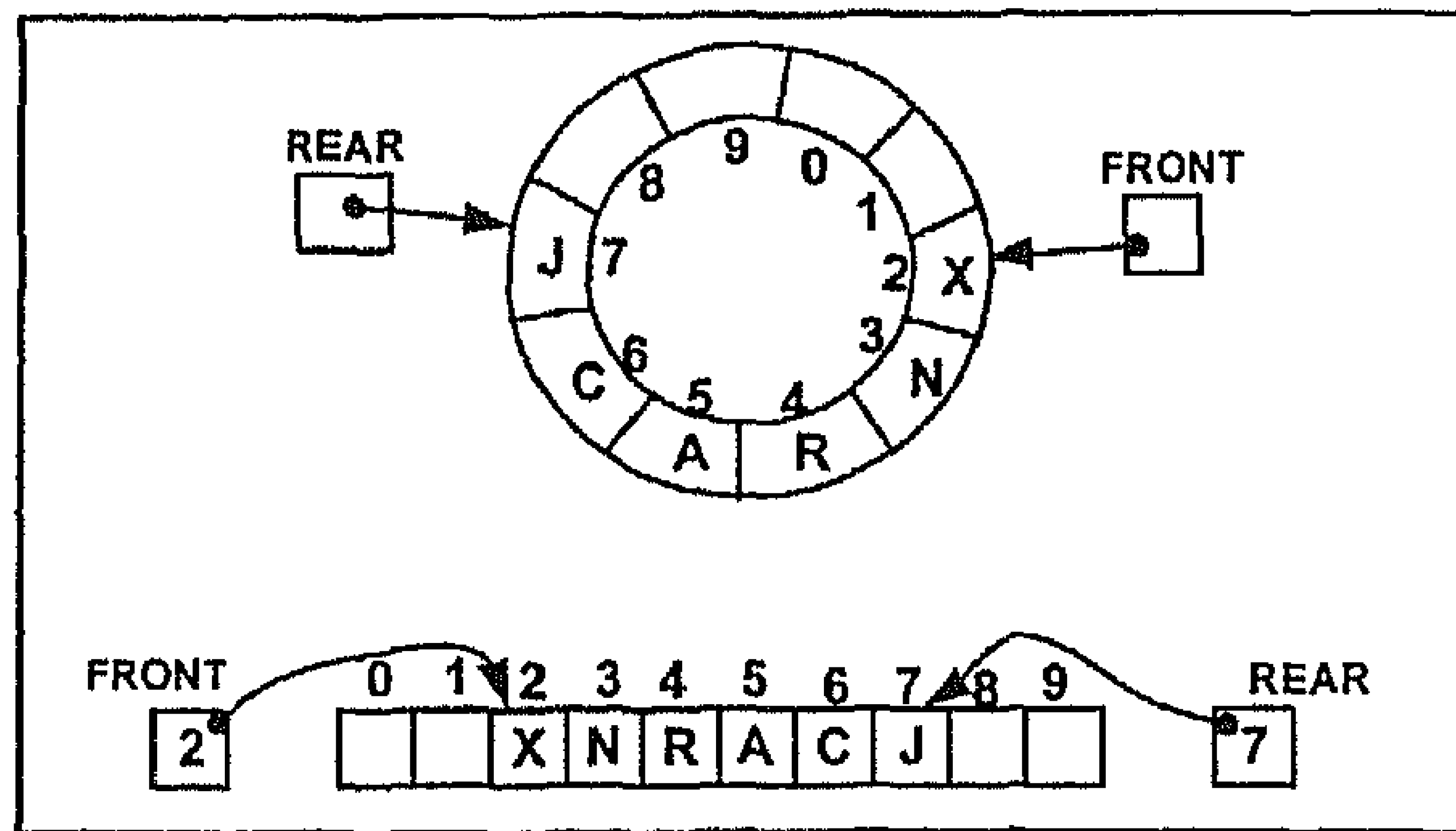
1. تتأثر عملية الإضافة إلى الطابور بعاملين أساسيين هما: ....
2. يمكن أن نصل إلى وضع الفائض (الامتلاء) (overflow) في حالة تمثيل الطوابير باستخدام ....
3. يتم تعديل قيمة المتغير (REAR) في كل حالة إضافة فعلية إلى الطابور الممثل على شكل مصفوفة، إلا في حالة أن تكون قيمة هذا المتغير مساوية لـ .... حيث يتم التعديل أيضاً على المتغير (FRONT).
4. عدد العناصر التي يمكن إضافتها إلى الطابور الممثل على شكل قائمة متصلة .... (محدود/غير محدود).
5. إذا كان العنصر المضاف إلى الطابور هو العنصر الوحيد في الطابور عند تلك اللحظة، فإن قيمة المتغيرين (REAR) و (FRONT) قبل إضافة هذا العنصر هي ....
6. يقتصر تأثير أسلوب تمثيل الطابور في الذاكرة فيما يتصل بعملية الحذف على ....
7. هناك ثلاثة أوضاع عامة لحذف العناصر من الطوابير هي ....
8. إذا كان الطابور يشتمل على عنصر وتم حذف هذا العنصر فإن قيمة كل من المتغيرين (REAR) و (FRONT) تصبح ....
9. يعبر عن الحذف من الطوابير باستخدام المصفوفات بالجملة التالية .... بينما يعبر باستخدام القوائم المتصلة بالجملة التالية ....
10. إذا كانت قيمة المتغير (FRONT) مساوية لقيمة المتغير (REAR) فإن ذلك يعني إما أن الطابور .... أو أن الطابور ....

5. الطوابير الدائرية وصيغ أخرى للطوابير

1.5 الطوابير الدائرية (Circular queues)

لا يوفر أسلوب تمثيل الطوابير باستخدام المصفوفات الأحادية وسيلة فعالة للتعامل مع الطوابير. ولعلك تذكر أننا اضطررنا إلى إجراء عملية إزاحة للقيم إلى بداية المصفوفة عندما أصبحت قيمة المتغير (REAR) مساوية للحد الأعلى للمصفوفة (1-) وكانت قيمة المتغير (FRONT) لا تشير إلى العنصر الأول في المصفوفة. وقد فعلنا ذلك من أجل توفير إمكانية الإضافة إلى الطابور. وكان بإمكاننا أن نفعل ذلك بأسلوب آخر وهو أن نقوم، في كل مرة تتم فيها عملية الحذف من الطابور، بإزاحة القيم إلى بداية المصفوفة على النحو الذي يتم على أرض الواقع في الحياة العملية.

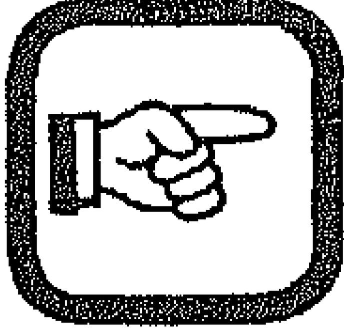
ولكن، سواء تمت عملية الإزاحة مرة واحدة، كما فعلنا في خوارزمية الإضافة (1)، أو تمت تدريجياً عند إجراء الحذف، فإن هذا الأسلوب لا يتسم بالكفاءة. ولا بد إذن، من البحث عن أسلوب آخر يتجنب فكرة الإزاحة مطلقاً، وهذا الأسلوب يتمثل في تطبيق فكرة الطابور الدائري. ومعنى ذلك أننا سننظر إلى المصفوفة من الناحية المنطقية على أنها تشكل حلقة متكاملة، يتصل آخرها بأولها وذلك على النحو الموضح في الشكل (11). ففي هذا الشكل تجد عرضاً للصورة الحقيقية للمصفوفة، وعرضاً آخر للتصور الدائري للمصفوفة نفسها وللقيم الممثلة للطابور.



الشكل (11) : التصور الدائري للطوابير الممثلة على شكل مصفوفات

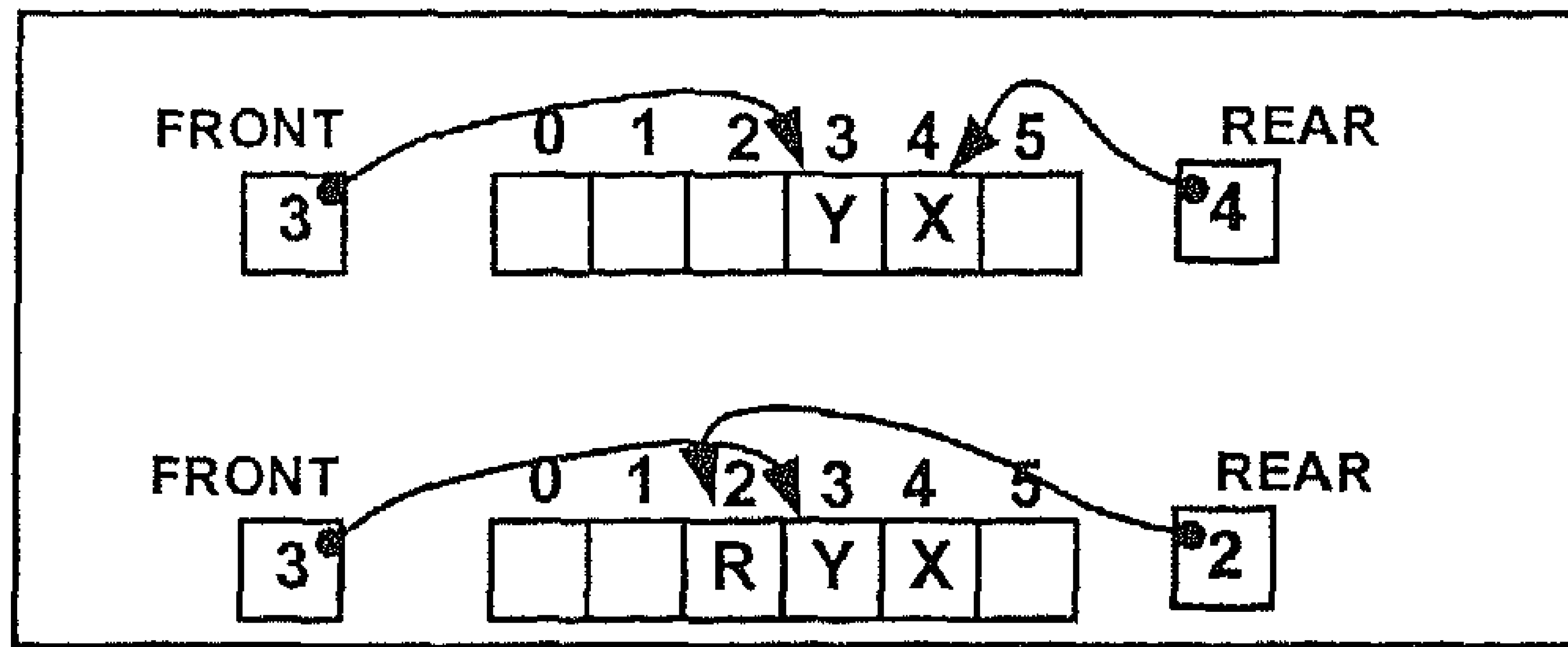
وبناءً على هذا المفهوم الجديد لأسلوب تمثيل الطابور باستخدام المصفوفات، فإن إضافة عنصر جديد إلى الطابور ستؤدي إلى تحريك المؤشر (REAR) خطوة واحدة إلى الأمام باتجاه حركة عقارب الساعة. كما أن حذف عنصر من الطابور سيؤدي إلى تحريك المؤشر (FRONT) خطوة مماثلة باتجاه عقارب الساعة إلى الموضع التالي.

وفي ضوء هذه الطبيعة الدائرية للطابور وحركة كل من المتغيرين (FRONT) و (REAR) باتجاه عقارب الساعة، فإننا سنواجه مشكلة تتعلق بتحديد الحالة التي يكون فيها الطابور خالياً من القيم. ومشكلة أخرى مماثلة تتعلق بتحديد الحالة يكون فيها الطابور مليئاً بالقيم أو ما أشرنا إليه من قبل بوضع الفائض. وسبب الإشكال يعود إلى أن قيمة المتغير (FRONT) قد تكون أكبر من قيمة المتغير (REAR) الأمر الذي لم نصادفه في تعاملنا مع الطوابير الاعتيادية التي تمت مناقشتها في القسم السابق. ولتوضيح هذه الفكرة دعنا نتأمل المثال التالي:

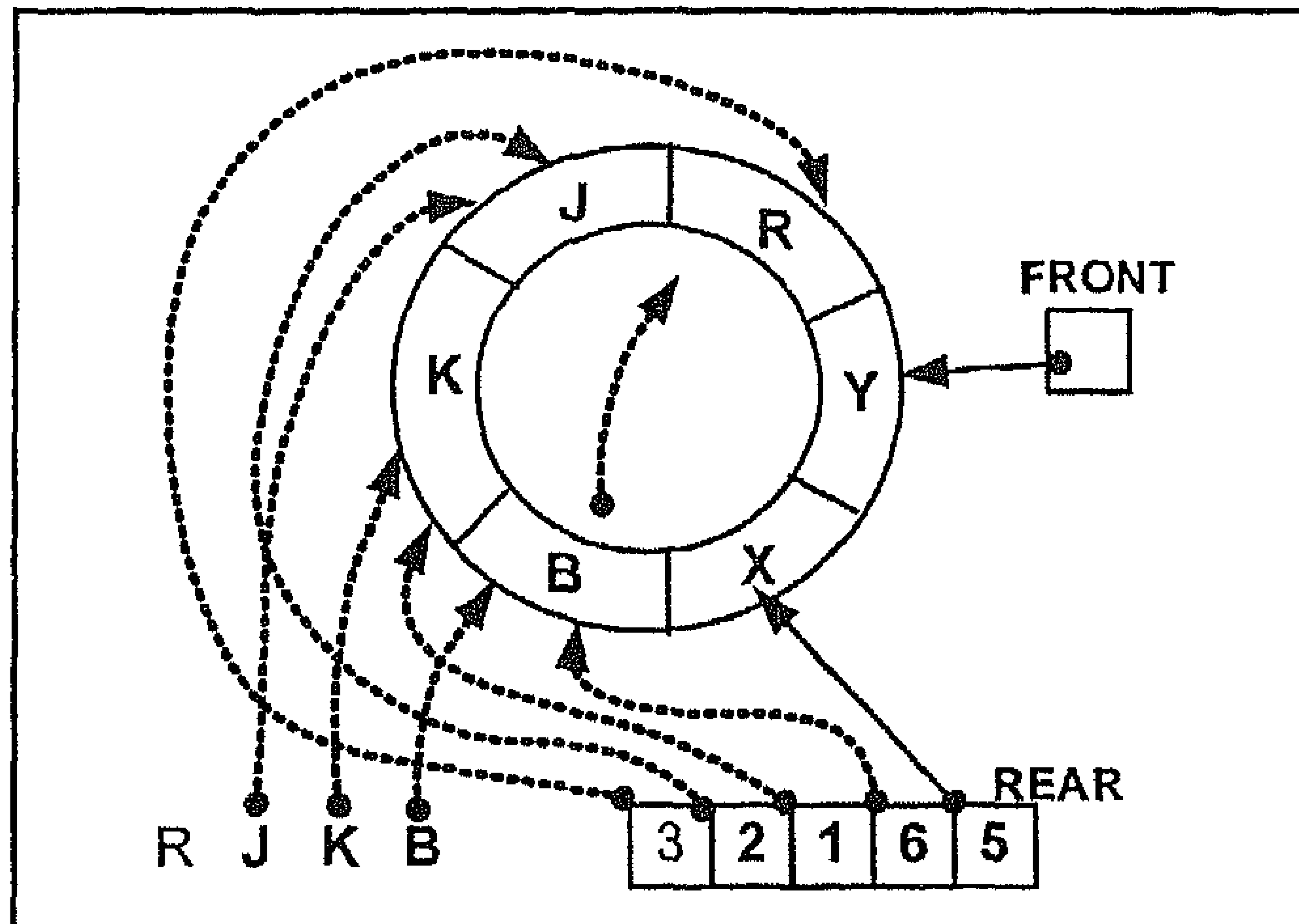


مثال (1)

افترض أن لدينا الطابور الموضح في الشكل (12 - أ)، وأننا نرغب في إضافة القيم التالية إلى الطابور على التتابع (من اليمين إلى اليسار): (R, J, K, B). فبعد إضافة القيمة الأولى المتمثلة بالقيمة B



الشكل (12 - أ): الطابور عندما تكون قيمة (REAR) أقل من قيمة (FRONT)

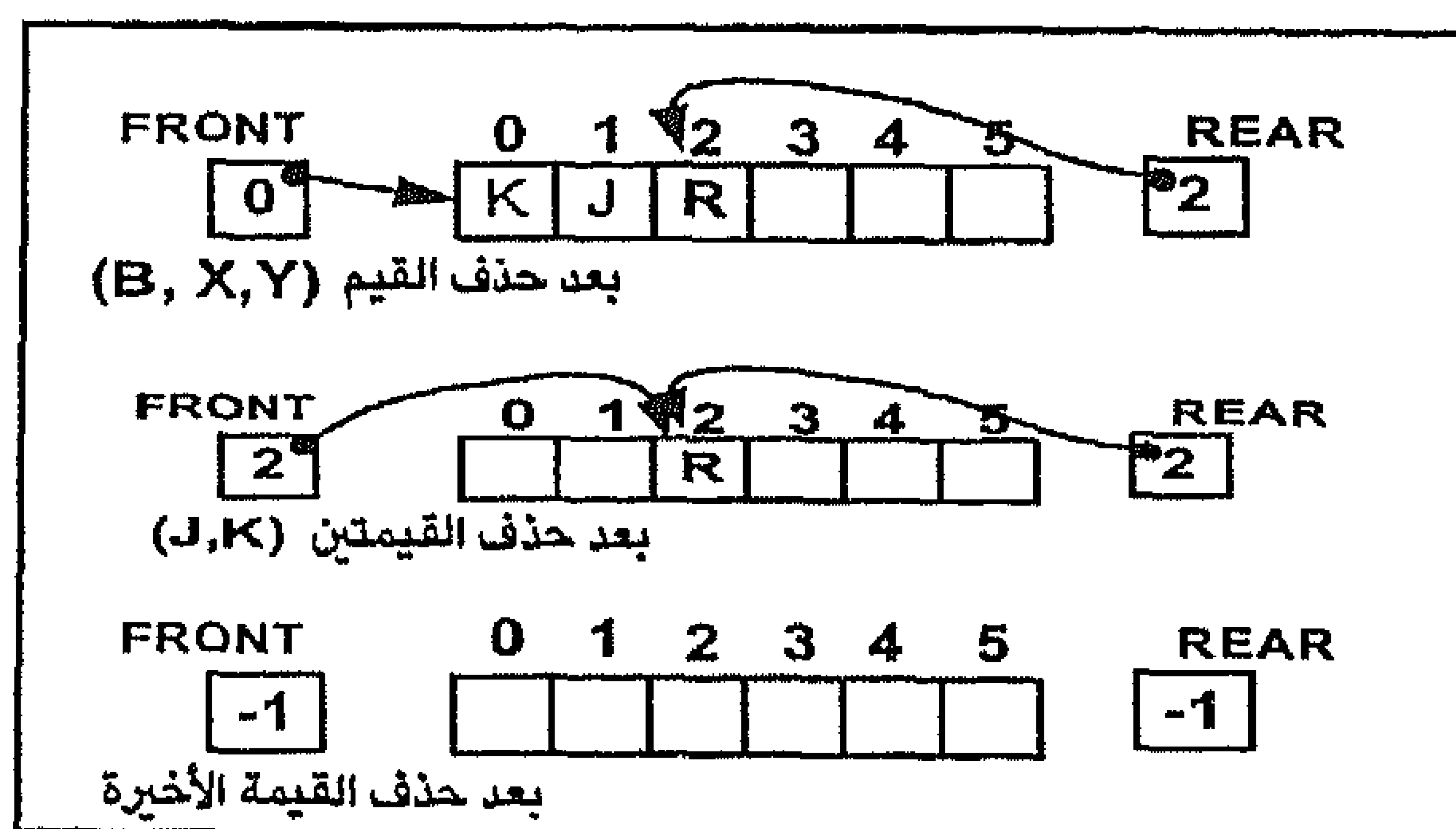


الشكل (12 - ب): الطابور الدائري خلال عملية الإضافة.

يصبح المؤشر (REAR) في وضع يشير فيه إلى هذا العنصر الجديد ورقمه "6". ثم نبدأ بإضافة العنصر الثاني المتمثل بالقيمة "K". وهنا نكتشف أننا وصلنا إلى نهاية المصفوفة حيث وصلت قيمة REAR إلى "6" . وبذلك نطبق فكرة الطابور الدائري، وتضاف القيمة الجديدة الثالثة وهي "J" إلى بداية المصفوفة حيث توجد أماكن شاغرة. ومعنى ذلك أن مؤخرة الطابور قد انتقلت إلى بداية المصفوفة وأصبح المتغير (REAR) في وضع يعكس هذه الحقيقة ونستمر بالإضافة حتى ننتهي من جميع القيم. وبذلك نصل إلى وضع أصبحت فيه المصفوفة مليئة بالقيم كما في الشكل (12 -ب)، أي أننا وصلنا إلى وضع الفائض (overflow) ولكن قيمة المتغير (REAR) أصبحت "3"، كما تلاحظ، وهي أقل من قيمة المتغير (FRONT) التي تساوي "4".

افترض الآن أننا نرغب في حذف العناصر الثلاثة الأولى من الطابور والمتمثلة بالقيم: (B, X, Y) فما الذي سيحدث؟ ان نتيجة الحذف هذه هي ان قيمة (FRONT) ستتغير مع كل عملية حذف. ففي المرة الأولى ستصبح قيمة هذا المؤشر خمسة، ثم تصبح ستة في المرة الثانية، ثم تصبح واحداً بعد حذف القيمة "B"، وذلك على النحو الموضح في الطابور الأول من الشكل (13). ولو قمنا أيضاً بحذف القيمتين التاليتين من الطابور لتبقّى لدينا في الطابور عنصر واحد فقط، هو ذلك العنصر المتمثل بالقيمة "R" كما هو واضح في الطابور الثاني من الشكل (13).

وبناءً على ذلك فقد تساوت قيمة المتغيرين (FRONT) و (REAR). ولو قمنا بحذف العنصر المتبقي في الطابور، فإن ذلك سينقل الطابور إلى وضع جديد تصبح فيه قيمة (FRONT) أكبر من قيمة (REAR) في ضوء الزيادة المضطردة التي تحدث لهذا المتغير نتيجة للحذف. ولكن، كما هو واضح، لم يبق في الطابور أي قيم. وينبغي في هذا الوضع الجديد أن يعكس المتغيران حقيقة خلو الطابور من القيم. ويمكن أن يتم هذا من خلال تعديل قيمة كل منهما إلى صفر.



الشكل (13): الطابور عندما تتساوى قيمة (FRONT) و (REAR)

وكما يتضح من هذا المثال فإن الطابور يمكن أن يصل إلى وضع الفائض أو الامتلاء (overflow) في حالتين هما:

أولاً: عندما تصبح قيمة (FRONT = 0) وقيمة (REAR = -1).

ثانياً: عندما تصبح قيمة (REAR= FRONT - 1).

أما الوضع الخالي (empty) فإنه يحدث في حالة واحدة فقط، وهي عندما تكون قيمة (FRONT=REAR=0) حيث يتم حذف العنصر الوحيد المتبقي في الطابور وتصبح قيمة (FRONT=0) وقيمة (REAR=0). وكما يتضح من المثال السابق أيضاً فإن الطبيعة الدائرية للطابور تنعكس من خلال قيمتي (REAR) و (FRONT)، وذلك في الحالتين التاليتين:

أولاً: عندما تكون قيمة (REAR = -1) وقيمة (FRONT > 0)

ثانياً: عندما تكون قيمة (FRONT = -1) وقيمة (MaxSize > REAR > 0) وفي كلتا الحالتين تصبح قيمة المتغير المعني بالحركة واحداً بعد أن كانت قيمته MaxSize.

هذا هو ما يميز الطوابير الدائرية عن الطوابير الاعتيادية التي سبقت مناقشتها، وفيما يلي نستعرض خوارزميتي الإضافة والحذف لهذه الصيغة في معالجة الطوابير.

الخوارزمية (5):

```
void MyQueue::EnQueue(MyQueue& Q, int element)
  {//inserting new values into an array-based circular queue
     int state;
      state= Q.REAR - Q.FRONT + 1;
      if (state==0 ||state==MaxSize)
      cout<<"queue is full .. overflow";
              else
                    {if (Q.FRONT== -1)  // queue is empty
              {Q.FRONT=0;
               Q.REAR=0;
               }
             else
               if (Q.REAR== -1) // move clockwise
                Q.REAR=0;
                else  //increment REAR
                            Q.REAR=Q.REAR + 1;
          Q.data[Q.REAR]=element; // insert
         }// end of insertion cases
    }
```

ولو نظرنا، عزيزي الدارس، إلى هذه الخوارزمية لوجدنا أن لدينا أربعة أوضاع مختلفة للإضافة، وهي:

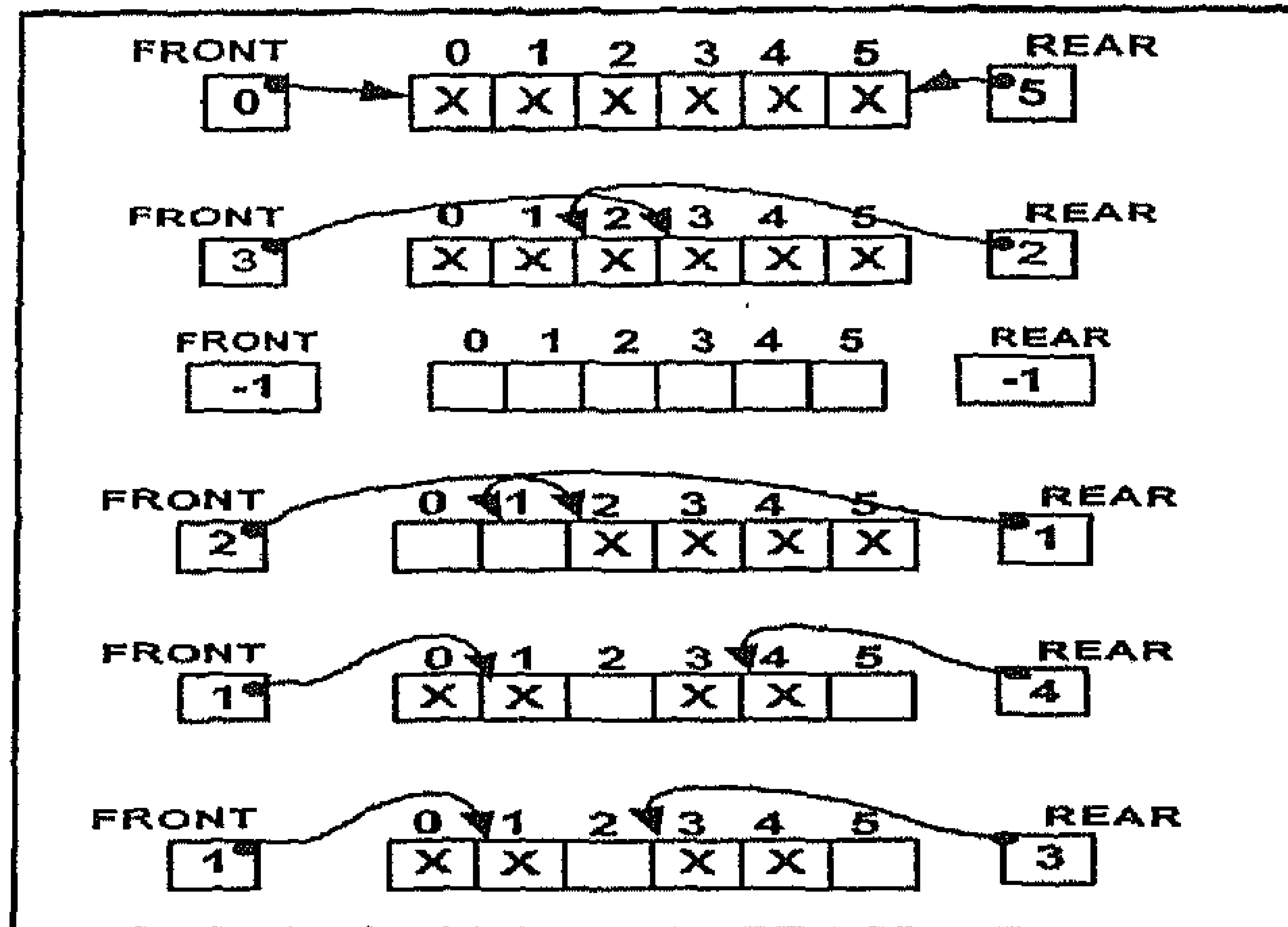
أولاً: الطابور ممتلئ، وبذلك فإنه ليس هناك مجال لإضافة أي قيم جديدة.

ثانياً: الطابور فارغ، وبذلك فإن العنصر المضاف هو الوحيد في الطابور.

ثالثاً: الطابور لم يعد يسمح بالإضافة إلى نهاية المصفوفة، وبذلك تتم الإضافة إلى بداية المصفوفة نتيجة لتطبيق فكرة الطابور الدائري.

رابعاً: الطابور في وضعه الاعتيادي، وبذلك تتم الإضافة في المكان المناسب.

والشكل (14) يعرض هذه الأوضاع الأربعة. وكما تلاحظ فإن الفرق بين هذه الخوارزمية والخوارزمية السابقة التي تعالج مسألة الحذف في الطوابير غير الدائرية يكمن في الوضعين الأول والثالث بشكل خاص.



الشكل (14): الأوضاع المختلفة للإضافة إلى الطابور الدائري

الخوارزمية (6):

```
void MyQueue::DeQueue(MyQueue& Q, int& element)
   {//deleting an element from an array-based circular queue
    if(Q.FRONT== -1)
                  cout<<"queue is empty... undeflow";
    else
          { element=Q.data[Q.FRONT];
       if (Q.FRONT==Q.REAR) // queue has one element
        {Q.FRONT= -1;
         Q.REAR= -1;
```

```
      }
      else
      if (Q.FRONT==MaxSize) //  move clockwise
      Q.FRONT=0;
      else
        Q.FRONT=Q.FRONT + 1;
   }    }
```

ولو تأملنا هذه الخوارزمية جيداً، لوجدنا أنها تعالج أربعة أوضاع مختلفة شأنها في ذلك شأن خوارزمية الإضافة وهذه الأوضاع هي:

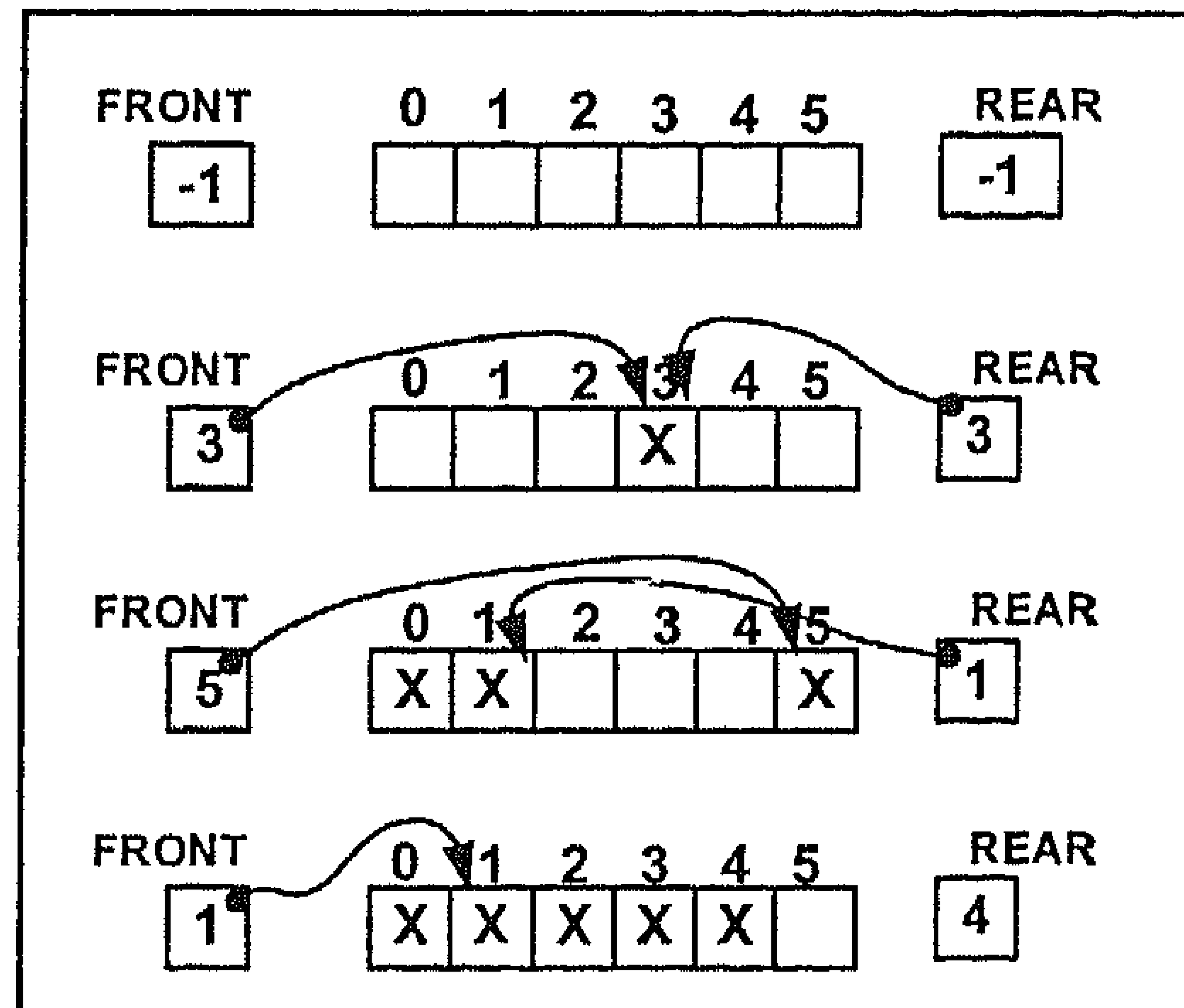
أولاً: الطابور فارغ، وبذلك فإنه ليس هناك حذف.

ثانياً: الطابور يتكون من عنصر واحد، وبذلك فإن الطابور يصبح خالياً من القيم بعد إجراء عملية الحذف.

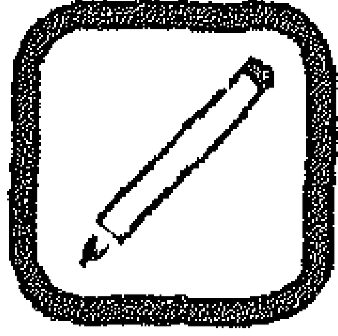
ثالثاً: الطابور مكون من مجموعة من القيم، أحدها في نهاية المصفوفة والأخرى موجودة في بداية المصفوفة، وبذلك يتحرك المؤشر باتجاه عقارب الساعة إلى بداية المصفوفة.

رابعاً: الطابور في وضعه الاعتيادي، وبذلك تتم عملية الحذف في المكان والأسلوب المناسبين.

والشكل (15) يعرض هذه الأوضاع الأربعة. وكما تلاحظ، فإن الفرق بين هذه الخوارزمية وخوارزمية الحذف التي تمت مناقشتها من قبل يكمن في الوضع الثالث حيث يتم التعبير عن الوضع الدائري للطابور.

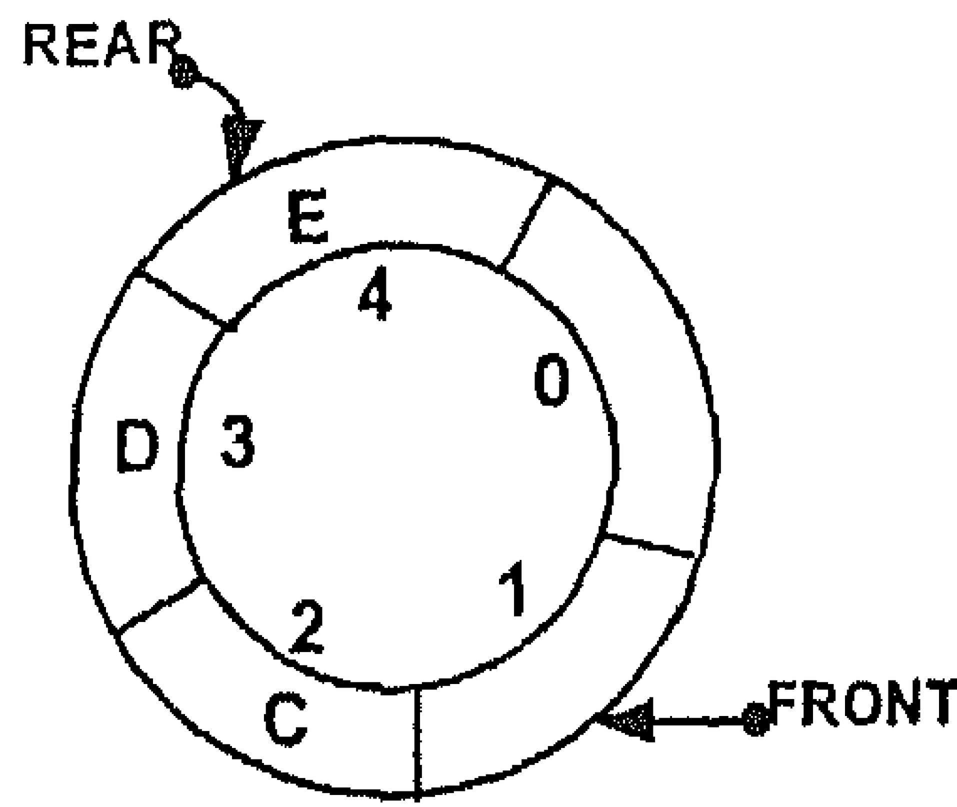


الشكل (15): الأوضاع المختلفة للحذف من الطابور



تدريب (5)

تأمل الطابور الدائري الموضح في الشكل (16) أدناه، ثم بين بالرسم وباستخدام الخوارزميتين المذكورتين أعلاه، ضمن برنامج تقوم بكتابته وتنفيذه على الجهاز، ما يحدث لهذا الطابور بعد إجراء عمليات الإضافة والحذف المذكورة في الجدول المصاحب والتي تحدث تباعاً كما هو محدد.



القيمة	العملية
F	إضافة
U.C	حذف
G	إضافة
E	حذف

الشكل (16): توضيح لعمليات الإضافة والحذف التي تتم على الطابور الدائري (تدريب (5))

2.5 طوابير الأولوية (Priority queues)

لقد عالجنا حتى الآن، عزيزي الدارس، الطوابير من وجهة النظر العامة المعروفة والقائمة على المبدأ القائل: "من يأتي أولاً يخدم أولاً". ولكن، في الحقيقة هناك بعض المشكلات والمواقف العملية التي تقتضي مخالفة هذا المبدأ إلى حد معين. هب نفسك، مثلاً، في طابور ينتظر إتمام إجراءات الجمارك في أحد المطارات وأنك لا تحمل في يدك إلا حقيبتك اليدوية التي تضم بعض أوراقك الشخصية. فهل من المناسب أن تبقى في هذا الطابور وتعامل معاملة الذين يحملون حقائب عديدة ومختلف أنواع الأمتعة والبضاعة؟ لا بد وأنك ستضيق ذرعاً بهذا الطابور وتعده إجراء غير مناسب وغير عادل. وربما لو طلب منك أن تنظّم أمر هذا الطابور لحاولت تجزئته حسب خطة معينة تأخذ بالاعتبار الزمن اللازم لإتمام عملية تفتيش الجمارك.

هذا الموقف نفسه يمكن أن يحدث في أسلوب تعامل جهاز الحاسوب مع البرامج والأعمال المختلفة التي تقدم له، للقيام بعمليات المعالجة التي تستلزمها. فالزمن الذي تحتاجه هذه الأعمال من وحدة الحساب والمنطق وموارد الجهاز الأخرى متفاوتة إلى حد

كبير. فبعضها قد يحتاج إلى دقائق، وبعضها الآخر يحتاج إلى جزء بسيط من الثانية. ومن أجل ذلك تلجأ نظم التشغيل إلى تبني أسلوب معين لتجاوز مشكلة تفاوت الوقت. ومن بين الأساليب المتبعة استخدام ما يسمى طابور الأولوية. ولكي نوضح فكرة هذه الصيغة من الطوابير، دعنا نورد المثال التالي:

مثال (2)

افترض أن مركز الحاسوب في إحدى الجامعات يقدم خدماته لأربع فئات من المستفيدين، هي: فئة الطلبة، وفئة المدرسين والباحثين، وفئة الإداريين، وفئة الجمهور الخارجي. وافترض أن سياسة تقديم خدمات مركز الحاسوب تقضي بتنظيم أولويات الإفادة من وقت الحاسوب على النحو التالي:

الفئة	الأولوية
الإدرايون	1
الباحثون والمدرسون	2
الطلبة	3
الجمهور الخارجي	حذف

معنى ذلك أن الإداريين يحظون بالأولوية العليا في التعامل، وأن الجمهور يتمتع بالأولوية الأخيرة.

وبناء على ذلك، فإن الأشخاص سيوزعون على هذه الفئات الأربعة وفقاً لنوع الأولوية التي يتمتعون بها. ويعبر عن هذه الأولوية برمز أو رقم يعكس وزن هذه الأولوية ضمن التسلسل المعتمد. وعليه دعنا نفترض أن لدينا الحالات التالية للتعامل معها:

الشخص	الأولوية	الشخص	الأولوية
B	2	X	1
R	1	K	4
S	4	M	3
J	2	D	2
N	3	I	4

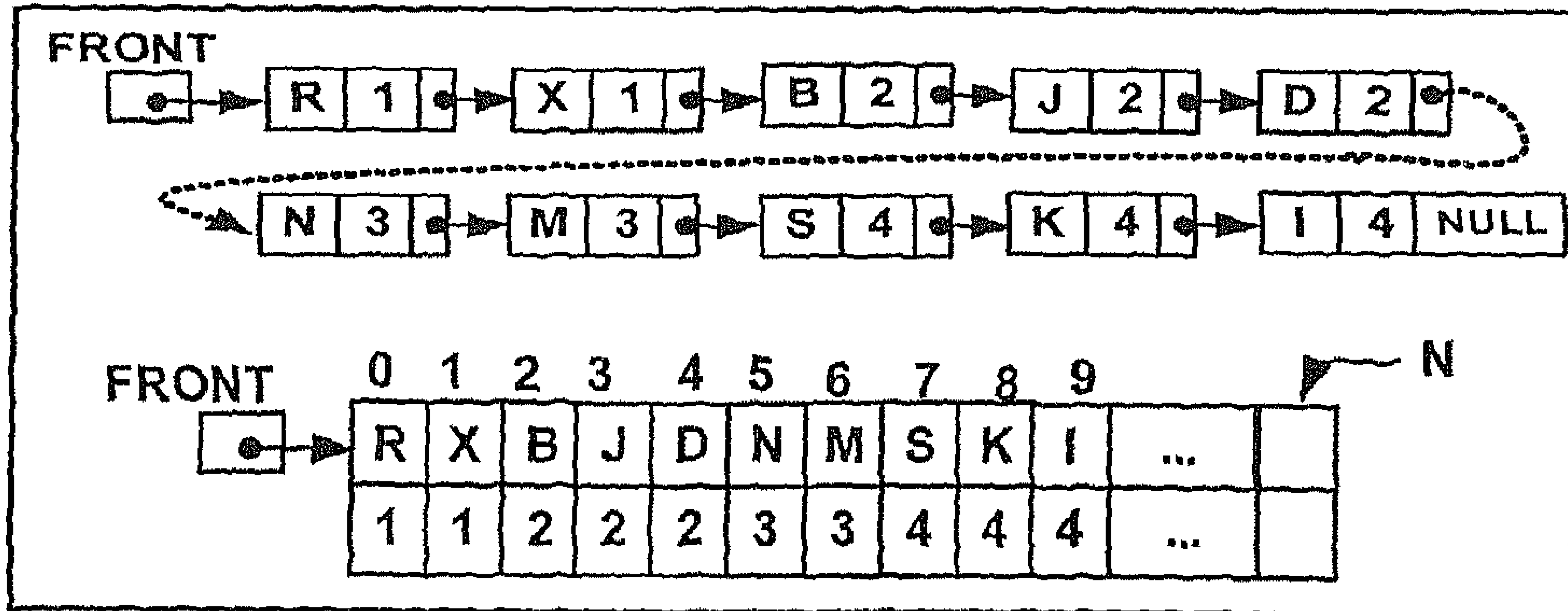
ولدى فرز هذه الحالات يتبين أن لدينا شخصين يتمتعان بالأولوية الأولى، وثلاثة أشخاص يتمتعون بالأولوية الثانية، وشخصين يتمتعان بالأولوية الثالثة، وثلاثة أشخاص يتمتعون بالأولية الأخيرة.

والمشكلة التي تبرز الآن هي كيف يمكن تنظيم الطابور نفسه داخل الذاكرة؟ هل نضعهم ضمن التسلسل الذي وردوا فيه؟ ولكن هذا لن يساعدنا كثيراً، لأننا سنضطر للبحث عن تسلسل الأولويات ضمن القائمة كلها. فهل نضع إذن كل فئة في قائمة منفصلة على حدة؟ إن هذا، بالطبع، أجدى. ولكننا مضطرون للتعامل مع قوائم مستقلة مما يستنفذ وقتاً طويلاً.

هناك، في الحقيقة، عدة أساليب لتمثيل هذه الصيغة من الطوابير، إلا أننا سنقتصر هنا على ذكر أسلوبين فقط:

الأسلوب الأول هو تمثيل الطابور على شكل قائمة خطية ملتزة (أي باستخدام مصفوفة ذات بعد واحد) أو متصلة (أي باستخدام قائمة متصلة)، بالصيغة الدائرية أو غير الدائرية. وفي هذه الحالة لن يحتاج الطابور إلا للمتغير (FRONT)، والسبب في ذلك أن الإضافة ستتم في المكان المناسب ضمن تسلسل الأولويات ولا يوجد حاجة إلى استخدام المتغير (REAR)، وإن تم استخدامه في حالة القوائم الملتزة فما ذلك إلا لغايات تحديد نهاية القائمة وليس لأغراض الإضافة بالتخصيص. أما الحذف فإنه سيتم من البداية بالأسلوب نفسه الذي تم اتباعه في الطوابير بصفة عامة، سواء أكانت دائرية أم غير دائرية. والشكل (17) يوضح هذا الأسلوب في تمثيل طوابير الأولوية.

ولعلك تلاحظ، عزيزي الدارس، بأن كل عنصر من العناصر قد تم التعبير عنه بسجل يتكون من ثلاثة حقول في حالة القوائم المتصلة، وهي: القيمة نفسها، ودرجة الأولوية، ومؤشر إلى العنصر التالي ومن حقلين في حالة القوائم الملتزة وهما: القيمة والأولوية.



الشكل (17): أسلوب تمثيل طوابير الأولوية باستخدام القوائم المتصلة والملتزة

وفيما يلي نعبر عن هذه الحقول في لغة سي++ بالتعريفات البرمجية التالية:

أولاً: طابور الأولوية كقائمة متصلة:

```
class QPtr
    {public:
     char INFO;
     int PRTY[4];
     QPtr * NEXT;  };
class MyQueue
 {public:     QPtr * FRONT;
                     QPtr * REAR;
              ...                  };
```

ثانياً: طابور الأولوية كقائمة ملتزة:

```
class MyQueue {
      private:
       char info;
       int PRTY [4], FRONT,REAR;
       int QUEUE[ MaxSize];
      public:
       ... };
```

وفي ضوء هذا المفهوم، فإن الاختلاف بين طوابير الأولوية والصيغ الأخرى للطوابير تكمن في أسلوب الإضافة. فبينما تتم عملية الحذف على النحو الذي وصفناه فيما سبق، فإن عملية الإضافة لا تختلف عن الأسلوب المتبع في إضافة عناصر جديدة إلى القوائم المتصلة أو القوائم الملتزة المرتبة تصاعدياً. وينبغي التنبيه هنا إلى أن العنصر المضاف ينبغي أن يأتي بعد كل العناصر الموجودة في الطابور التي تحمل درجة الأهمية نفسها في حالة تشابه قيمة الأولوية. وبناءً على ذلك فإننا لو أردنا إضافة العنصر المتمثل بالقيمة "E" والذي يتمتع بأولوية من الدرجة الثانية، فإن ذلك يعني أن هذا العنصر الجديد سيضاف بعد العنصر المتمثل بالقيمة "D" مباشرة. أي أننا نطبق المبدأ العام الخاص بالطوابير ولكن ضمن عناصر درجة الأولوية الواحدة.

هذا هو أحد أساليب تمثيل الطوابير في ذاكرة الحاسوب، والأسلوب الثاني هو استخدام المصفوفات الثنائية. وفي هذه الحالة يخصص لكل فئة من فئات الأولوية صف مستقل. كما يمكن النظر إلى هذا الصف على أنه يشكل قائمة مستقلة : دائرية أو غير دائرية حسب ما ترغب.

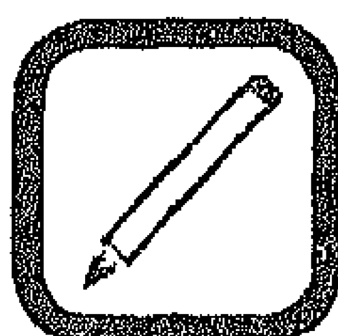
وبذلك فإنه يتم تخصيص متغيرين لكل صف شأنه في ذلك شأن الطوابير المستقلة،

أحدهما للإشارة إلى بداية طابور ذلك الصف وهو (FRONT) والآخر للإشارة إلى مؤخرة طابور ذلك الصف وهو (REAR). والشكل (18) يوضح هذا النوع من التمثيل.

	FRONT	0	1	2	3	4	5	6	REAR
0	1	R	X						1
1	1	B	J	D					2
2	1	N	M						1
3	1	S	K	I					2

الشكل (18): طريقة تمثيل طابور الأولوية باستخدام المصفوفات الثنائية

وبناء على هذا الأسلوب في تمثيل طوابير الأولوية نستطيع أن نقوم بعمليتي الإضافة والحذف على كل طابور من هذه الطوابير الأربعة بالأسلوب نفسه الذي شرحناه سابقاً عند الحديث عن الطوابير الدائرية. ولكن ينبغي أن نراعي أن الإضافة يمكن أن تتم في أي صف من الصفوف الأربعة الموضحة في الشكل وفقاً لدرجة الأولوية، إلا أن الحذف ينبغي أن يبدأ بالقيم الموجودة في الصف الأول، متلوة بالقيم الموجودة في الصف الثاني... وهكذا على التتابع. ومعنى ذلك أنه إذا قمنا بحذف القيم الموجودة في الصف الأول والقيمة الأولى في الصف الثاني، ثم تلا ذلك مباشرة عملية إضافة عنصر جديد إلى الصف الأوّل ومن ثم تلته مباشرة عملية حذف، فإن عملية الحذف هذه ستتناول العنصر الذي تمت إضافته حديثاً إلى الصف الأول وليس العنصر الثاني في الصف الثاني.

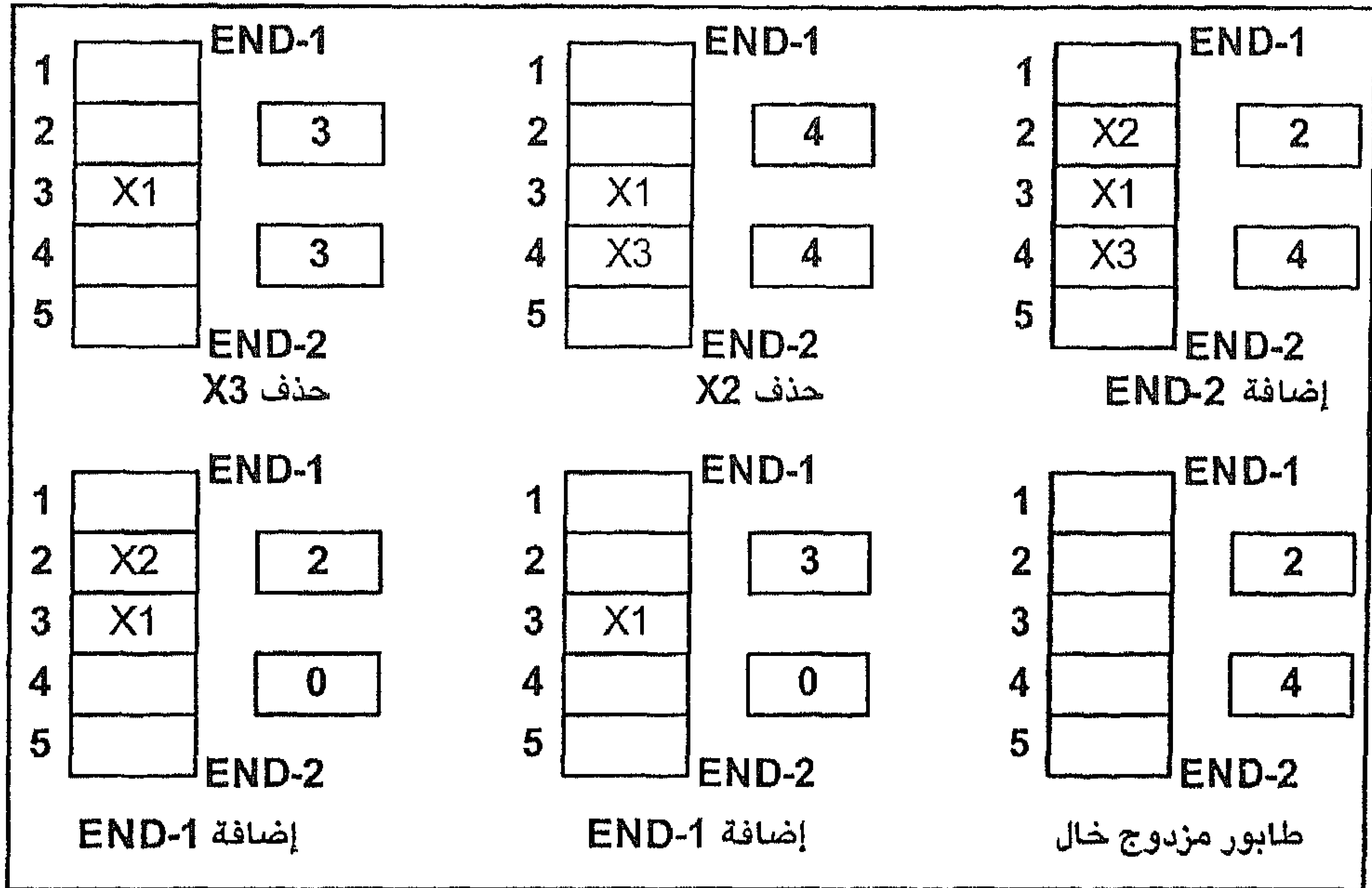


تدريب (6)

مستفيداً من الخوارزميتين (5، 6) الخاصتين بعمليتي الإضافة والحذف في الطوابير الدائرية، اكتب برنامجاً بلغة سي++ لإجراء عمليتي الإضافة والحذف في طوابير الأولوية وفق أسلوب التمثيل الموضح في الشكل (18).

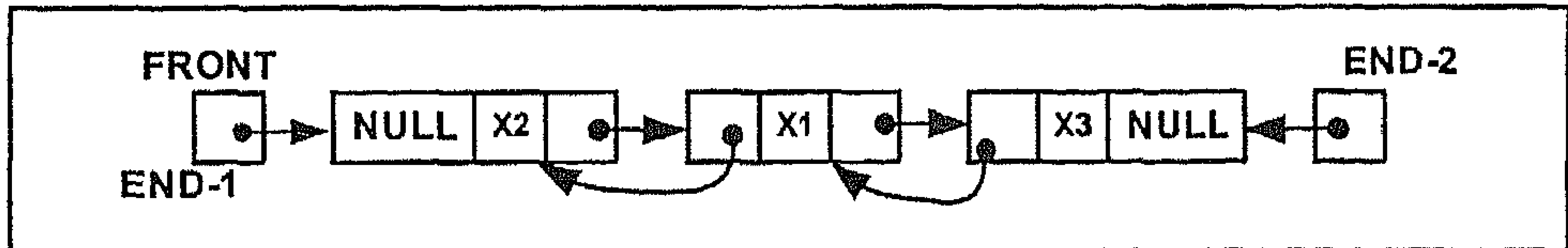
3.5 الطوابير المزدوجة (double-ended queues: deque)

الطوابير المزدوجة صيغة عامة أخرى للطوابير إلا أنها، تسمح بالإضافة والحذف من كلا الطرفين. فلو تأملنا الشكل (19)، لوجدنا أن بإمكاننا أن نقوم بالإضافة والحذف معاً باتجاه نهايته الأولى المعبر عنها بالمتغير (END-1) وباتجاه نهايته الأخرى المعبر عنها بالمتغير (END-2). وهو بهذا المعنى أشبه ما يكون بمكدسين تم إلصاق مؤخرة الواحد منهما بمؤخرة الآخر.

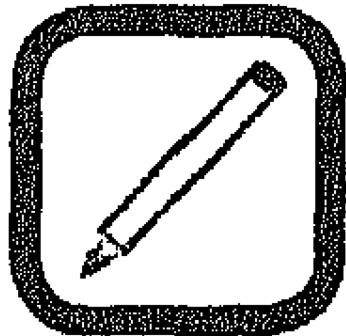


الشكل (19): طابور مزدوج في أوضاع مختلفة للإضافة والحذف

إن من الممكن تمثيل الطوابير المزدوجة باستخدام المصفوفات، ويمكن أيضاً تمثيلها على شكل قوائم متصلة ثنائية وذلك على النحو الموضح في الشكل (20).



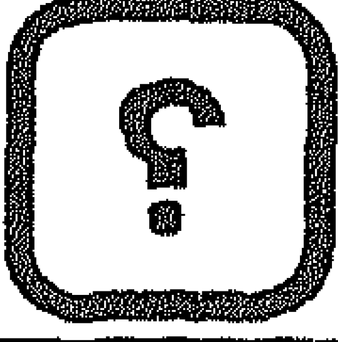
الشكل (20): تمثيل الطابور المزدوج باستخدام القوائم المتصلة الثنائية.



تدريب (7)

تأمل الشكل (20) مرة أخرى، واكتب الجمل اللازمة لإضافة أحد العناصر إلى (END-2)، وأربعة جمل أخرى لحذف أحد العناصر من جانب المتغير (END-1).

أسئلة التقويم الذاتي (4)



أجب بنعم أو لا:

1. الهدف الأساسي من تطبيق فكرة الطابور الدائري هو تجنب عملية الإزاحة الناتجة عن محاولة توفير حيز للإضافة في نهاية المصفوفة.
2. تصلح فكرة الطابور الدائري بصفة خاصة في التمثيل القائم على استخدام القوائم المتصلة.
3. الطابور الدائري يتخذ شكل حلقة كاملة يتصل أولها بآخرها داخل ذاكرة الحاسوب.
4. عندما تكون قيمة (REAR=N) و (FRONT=1) فإن الطابور يكون في وضع الفائض.
5. عندما تكون قيمة (REAR=FRONT+1) فإن الطابور يكون في وضع الفائض (ممتلئ).
6. عندما تكون قيمة (REAR=1) و (FRONT=N) فإن ذلك يعني أن الطابور خال من القيم.
7. يتحرك كل من المتغيرين (REAR) و (FRONT) في حالة الطوابير الدائرية باتجاه عقارب الساعة.
8. يسمح طابور الأولوية بالحذف من نهايته باستخدام المتغير (REAR).
9. يمكن تمثيل طابور الأولوية باستخدام صيغة القوائم المتصلة.
10. الطوابير المزدوجة تتيح المجال للإضافة والحذف من جانب واحد وهو مؤخرة الطابور.

6. الخلاصة

ناقشنا في هذه الوحدة نوع هام من التراكيب البيانية الخطية وهو الطابور. وهو تركيب بياني يستخدم كثيراً عندما نحتاج إلى نوع خاص من القوائم تتم الإضافة فيه إلى جانب يدعى المؤخرة REAR والحذف (أو الخدمة) من الجانب الآخر الذي يدعى المقدمة FRONT. وعليه فإن ما يأتي أولاً للطابور يخدم أولاً ويدعى هذا النمط فيفو FIFO (First In First Out) ومن الممكن تمثيل الطابور إما باستخدام المصفوفات (طريقة تتابعية) أو باستخدام القوائم المتصلة (التمثيل المتصل). والعمليات المستخدمة عادة على الطوابير هي:

1. إنشاء الطابور QueueCreate.
2. فحص فيما إذا كان الطابور خالياً (فارغاً) QueueEmpty.
3. فحص فيما إذا كان الطابور ممتلئاً (فحص الفائض) Queuefull.
4. إضافة عنصراً إلى الطابور Enqueue.
5. حذف عنصر من الطابور Dequeue.

ثم ناقشنا ثلاثة أنواع خاصة من الطوابير الأول يستخدم المصفوفة بطريقة دائرية وعليه فهو يدعى بالطابور الدائري ويهدف إلى تخليصنا من الحاجة إلى إزاحة عناصر الطابور من حين إلى آخر كي نتمكن من الإضافة. أما النوع الخاص الثاني فهو ما يدعى بطابور الأولوية حيث يستخدم حينما نحتاج إلى إعطاء بعض العناصر أولوية في الخدمة على غيرها من العناصر الأخرى. والنوع الثالث والأخير هو ما يدعى بالطابور المزدوج Deque حيث تتم الإضافة أو الحذف من أي جانب.

7. لمحة عن الوحدة الدراسية السابعة

موضوع الوحدة التالية هو المكدسات (Stacks). وهي تراكيب بيانية شبيهة إلى حد كبير بالطوابير، إلا أن عملية الإضافة والحذف تتم على نفس الجهة (أو الطرف). وعليه فإن أساليب الإفادة منها في تخزين البيانات ومعالجتها تختلف نوعاً ما عن الطوابير. وسنناقش في هذه الوحدة مسائل مثل أساليب تمثيل هذه التراكيب البيانية في الذاكرة وطرق بنائها وإجراء العمليات المختلفة عليها. وسنناقش الأنماط المختلفة التي ترد فيها والتطبيقات المختلفة التي تستخدمها.

8. إجابات التدريبات

تدريب (1)

1. التمثيل التتابعي

```
typedef const int myMaxQue;
myMaxQue MaxSize=10;
typedef char name[20];
 class MyQueue {
   private:
       name Q[ MaxSize];
       int FRONT,REAR;
};
```

2. التمثيل المتصل

```
class QPtr
    {public:
      char name[20];
      QPtr* next;
  };
class MyQueue
 {public:         QPtr* FRONT;
                        QPtr* REAR;};
```

تدريب (2)

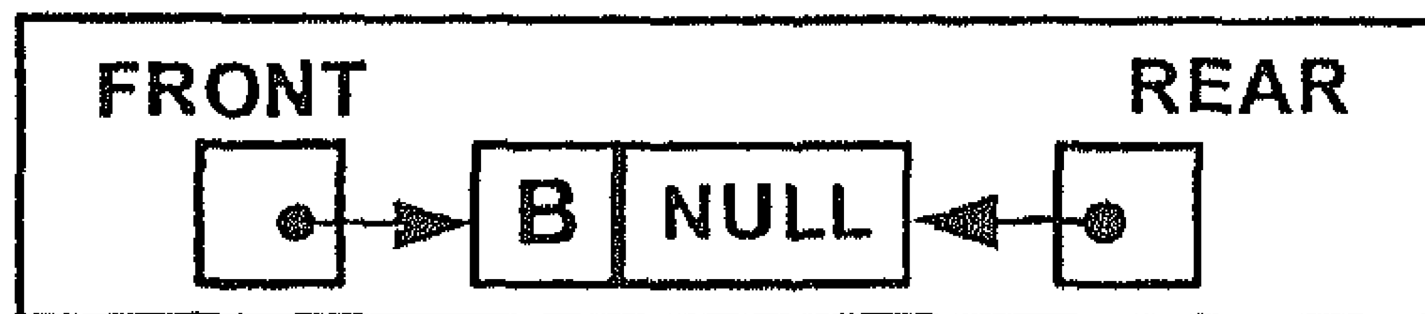
أ. الوضع الذي تمثله الحالة الأولى هو الوضع الاعتيادي الذي يتمثل في تنفيذ الجملتين الأخيرتين في الخوارزمية فحسب.

ب. تمثل الحالة الثانية الوضع المتمثل بـ: (REAR = N) حيث لا بد ن إزاحة القيم الموجودة إلى بداية المصفوفة لإتاحة المجال لإضافة العناصر الجديدة.

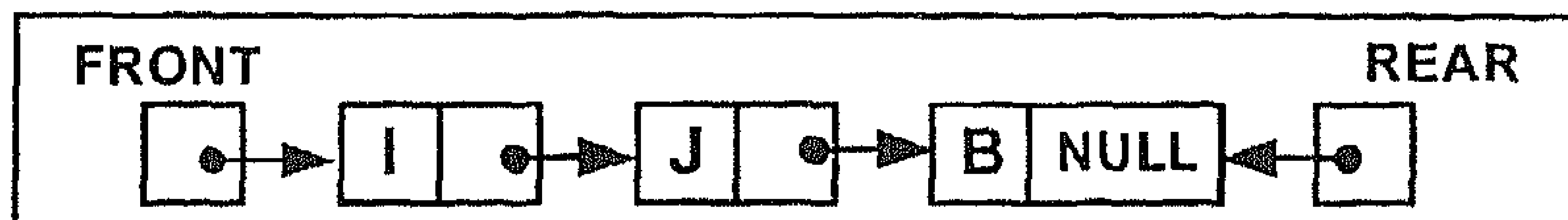
ج. تمثل الحالة الثالثة وضع الفائض بالنسبة للطابور. وبذلك فإنه لا ينفذ من الخوارزمية سوى الشرط الأول وجوابه.

د. تمثل الحالة الرابعة وضع الخالي (empty) بالنسبة للطابور، وبذلك فإن العنصر المضاف سيكون الوحيد في الطابور.

تدريب (3)



FRONT=newNode;
REAR =newNode;
REAR->next= newNode;
REAR= newNode;

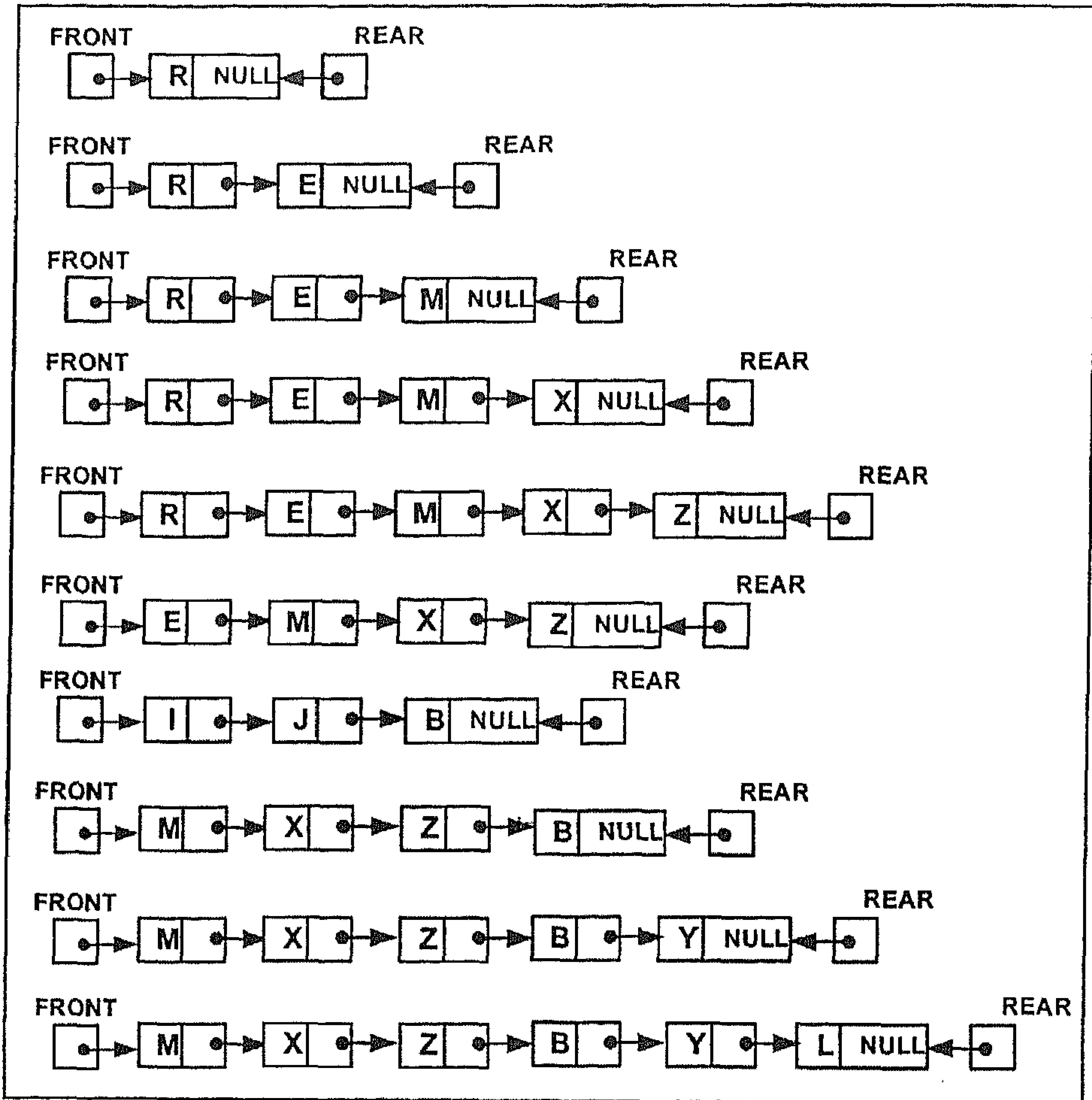


تدريب (4)

• باستخدام المصفوفات

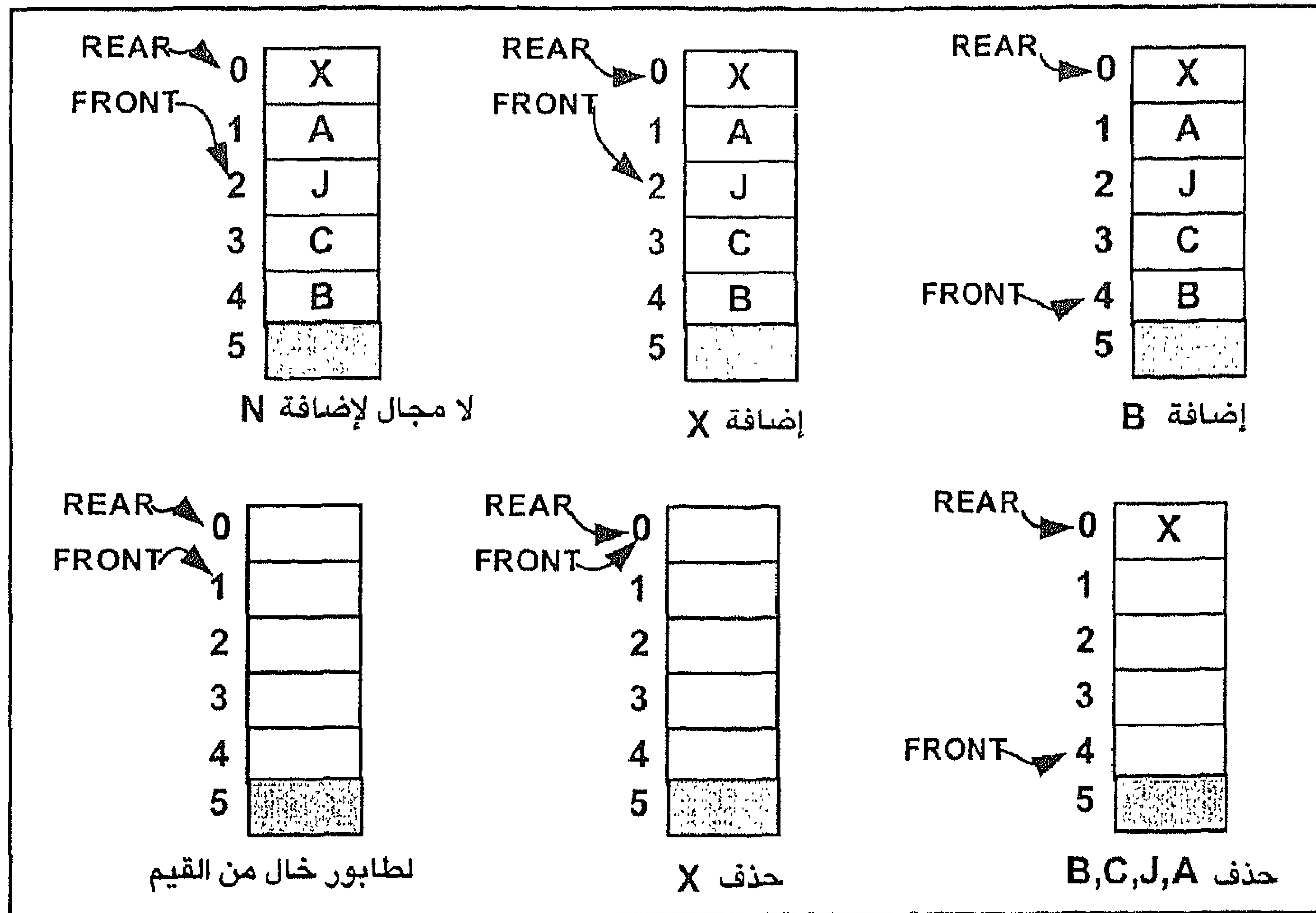
	1	2	3	4	5	6	REAR	
1	R						1	إضافة القيمة “R”
1	R	E					2	إضافة القيمة “E”
1	R	E	M				3	إضافة القيمة “M”
1	R	E	M	X			4	إضافة القيمة “X”
1	R	E	M	X	Z		5	إضافة القيمة “Z”
2		E	M	X	Z		5	إضافة القيمة “R”
3			M	X	Z		5	إضافة القيمة “E”
3			M	X	Z	B	6	إضافة القيمة “B”
1	M	X	Z	B	Y		5	إضافة القيمة “Y”
1	M	X	Z	B	Y	L	6	إضافة القيمة “L”
1	Z						1	إضافة القيمة “Z”

• باستخدام القوائم المتصلة



تدريب (5)

	حذف F		أضف E		أضف D,C		أضف D,C	
1	F	FRONT=1			F	REAR=1	F	REAR=1
2	G	REAR=2	G	REAR=2				
3							C	FRONT=3
4							D	
5			E	FRONT=5	E	FRONT=5	E	



تدريب (6)

```
typedef const int myMaxQue;
myMaxQue MaxSize=12;
typedef char XType;
const M=4, N=7;
typedef  XType  QUEUE[N];
typedef  XType  TABLE[M];
typedef  int index[M];
 class MyQueue {
  private:
    int data [ MaxSize];
    int Len,length; //pointer to last element
    int FRONT,REAR;
  public:
      void QueueCreate(MyQueue& Q);
      bool QueueFull(MyQueue Q);
      bool QueueEmpty(MyQueue Q);
      void CQINSERT (MyQueue& Q,  char ch);
      void  CQDELETE(MyQueue& Q, char& element);
   void  PrintQueue(MyQueue Q);
};
main()
  {MyQueue Que;
```

```
struct QStrct
  {char info;    int PRTY;}R;
TABLE T;   QUEUE Q;     index AFRONT, AREAR;
char PROCESS,ch;                  int  i,j;
cout<<"print the letter I for insertion, or D for deletion"<<endl;;
cin>>PROCESS;
if (PROCESS=='I')
 {cin>>R.info;
  do     cin>>R.PRTY; while (R.PRTY<1 || R.PRTY>4);
  i=R.PRTY;
  for (j=1;j<=N;j++)
   Q[j]= T[i,j];
  Que.CQINSERT(Que,R.info);
  j= AREAR [i];
  T[i,j]=Q [j];
 }
 else
  {i= 1;
   while (AFRONT [i]== 0 && i<=M)
    {i++;
     if  (i<= M)
      { for (j=1;j<=N;j++)
          Q[j]=T[i,j];
        Que.CQDELETE (Que,ch); }
       else
        cout<<"the priorty queue is empty";
    }
  } }
```

تدريب (7)

أ - الإضافة إلى "END2":

1. END2-> FORW = NEWNODE;
2. NEWNODE -> BACKW = END2;
3. END2 = NEWNODE;

ب - الحذف من "END1" :

1. PTR = END1;
2. END1 = END1-> FORW;
3. END1 -> BACK = NULL;
4. delete (PTR);

9. مسرد المصطلحات

- **الإضافة (Enqueue)**: مصطلح يشير إلى عملية الإضافة إلى الطوابير.
- **الحذف (Dequeue)**: مصطلح يشير إلى عملية الحذف من الطوابير.
- **طابور الأولوية (Priority queue)**: نوع من الطوابير يقوم على إضافة العناصر الجديدة إلى الطابور في المكان المناسب وفقاً لدرجة الأولوية، وليس في آخر الطابور كما هو معتاد.
- **الطابور الدائري (Circular queue)**: تصور منطقي للطابور يقوم على النظر إلى الطابور في حالتي الإضافة والحذف على أنه يشكل حلقة متكاملة يتصل أولها بآخرها.
- **الطابور المزدوج (Deque)**: نوع من أنواع الطوابير يسمح بالإضافة والحذف في كلا الاتجاهين، وبذلك فهو أقرب إلى المكدس منه إلى الطابور.
- **المقدمة (Front)**: مؤشر خاص يشير إلى العنصر الأول في الطابور، وهو يحدد اتجاه الحذف من الطابور.
- **المؤخرة (Rear)**: مؤشر خاص يشير إلى العنصر الأخير في الطابور، وهو يحدد اتجاه الإضافة إلى الطابور.



10. المراجع

1. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice- Hall. 2000.
2. Tremplay, J.P.; and Sorenson, P.G.; An Introduction to Data Structures with Applications, 2nd Edition., McGraw-Hill, 1984.
3. Amsbury, Wagne, Data Structures From Arrays to Priority Queues. Belmont (USA): Wadsworth, 1985.
4. Kruse, Robert L., Data Structures and Program Design. Englewood Cliffs (USA): Prentice-Hall, 1984.
5. Lipschutz, Seymour, Theory and Problems of Data Structures. New York: McGraw-Hill, 1986.
6. Miller, Lawrence H., Advanced Programming Design and Structures. New York: McGraw-Hill, 1986.
7. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, , Addison Wesley, 1999.
8. Lewis, T. G.; Smith, M.Z., Applying Data Structures. Atlanta (USA): Houghton Mifflin, 1976.
9. Tenenbaum, Aarom M.; Augenstein, Moshe J., Data Structures Using Pascal. Englewood Cliffs (USA): Prentice-HALL, 1981.



المكدسات Stacks

محتويات الوحدة

1. المقدمة

1.1 تمهيد

أهلاً بك، عزيزي الدارس، إلى الوحدة السابعة من كتاب "تركيب البيانات وتصميم الخوارزميات" وهي بعنوان "المكدسات".

لقد قدمنا لك، عزيزي الدارس، الطوابير في الوحدة السابقة وقد بينا أن الطوابير هي نوع خاص من القوائم يتم فيها الإضافة إلى مقدمة الطابور وتتم عملية الحذف من مؤخرة الطابور. أما في هذه الوحدة، عزيزي الدارس، فإننا سنقدم لك نوع جديد من تراكيب البيانات لا يقل أهمية عن الطوابير وهو المكدسات. والمكدس هو أيضاً نوع خاص من القوائم حيث تتم الإضافة والحذف من نفس الجهة. وسنبين في هذه الوحدة كيفية تمثيل وتنفيذ المكدسات والعمليات الخاصة بهم كما نبين أيضاً أهم التطبيقات التي نستخدم فيها المكدسات.

2.1 أهداف الوحدة

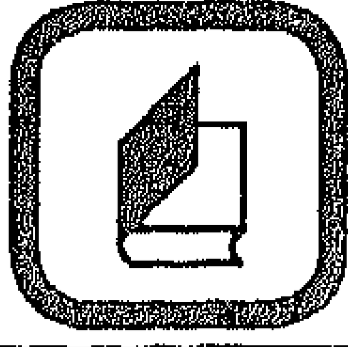
ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1. تبين مفهوم المكدسات والعمليات الخاصة بها.
2. تمثل وتنفذ المكدسات.
3. تستخدم المكدسات بطريقة صحيحة.
4. تبين بعض التطبيقات التي تستخدم المكدسات.

3.1 أقسام الوحدة

تتكون هذه الوحدة من ثلاثة أقسام رئيسة ترتبط بقائمة الأهداف السابقة. في القسم الأول ستعرض الوحدة مفهوم المكدسات ولمحة عن أهم استخداماتها ويرتبط هذا القسم بتحقيق الهدف الأول، وفي القسم الثاني ستعرض العمليات الخاصة بالمكدسات وكيفية تنفيذها ويرتبط هذا القسم بتحقيق الهدفين الأول والثاني. أما القسم الثالث فيهدف إلى تحقيق الهدفين الثالث والرابع عن طريق عرض مفصل لأهم تطبيقات المكدسات.

4.1 القراءات المساعدة



على الرغم من شمولية الأقسام والمسائل التي تم عرضها في هذه الوحدة إلا أن هناك فوائد كثيرة يمكن أن يجنيها الدارس من الرجوع إلى بعض القراءات الإضافية المساعدة. فهناك المزيد من المعلومات والأمثلة في هذه العصادر، وهناك أيضاً المزيد من التدريبات والأسئلة. ونوصي بالمصدرين التاليين لهذه الغاية، مع التذكير بأن قائمة المراجع في نهاية هذه الوحدة تتضمن مصادر على قدر كبير من الأهمية والفائدة:

1. Amsbury, Wayne, Data Structures from Arrays to Priority Queues. Belmont (USA): Wadsworth, 1985. pp. 97-119, 120-141, & 169-198.
2. D.S. Malik, Data Structures Using C++, 1st Edition, Course Technology, Inc., 2003.
3. Michael Main, Data Structures & Other Objects Using C++, 3d Edition, Addison-Wesley,2004

2. المكدسات وأسلوب تمثيلها واستخداماتها

إن المبدأ العام الذي يقوم عليه المكدس، عزيزي الدارس، هو أن الإضافة والحذف يجريان عند طرف واحد. وعندما نتحدث عن المكدسات في هذا السياق، فإننا نتحدث عن تركيب بياني يحاكي السلوك الواقعي ويقوم على الأساس نفسه. فإن المكدس ما هو إلا قائمة من القيم مخزنة بأسلوب معين من أساليب تمثيل القوائم وتعمل وفق مبدأ: "من يأتي آخراً يغادر أولاً". ومعنى ذلك أنّ آخر قيمة تضاف إلى المكدس هي أول قيمة تحذف منه. ويوصف هذا السلوك بالكلمة ليفو (LIFO) وتتكون من الأحرف الأولى للتعبير Last In First Out، وفي ضوء هذا المفهوم، فإن للمكدس بعض الخصائص التي يتسم بها في البناء وفي العمل. وأهم هذه الخصائص:

أ. يتسم المكدس بالديناميكية والحركة، شأنه في ذلك شأن الطوابير. فعلى الرغم من أن المستفيد لا يرى من المكدس إلا قمته أو سطحه العلوي فإن المكدس يزداد ويتقلص حسب الرغبة.

ب. يوفر المكدس إمكانية الوصول المباشر، إلا أن هذا الوصول محدود بعنصر واحد وهو قمة المكدس. كما أن الوصول إلى العناصر الأخرى مرهون بعمليات الإضافة والحذف أو ما نطلق عليه اصطلاحاً في هذا المقام الدفع والإطلاق (push and pop). فكلما زادت عمليات الحذف اقترب العنصر من القمة وكلما زادت عمليات الإضافة ابتعد العنصر عن القمة.

ج. إذا تساوت عمليات الإضافة وعمليات الحذف فإن المكدس يصل إلى وضع يصبح معه خالياً من القيم.

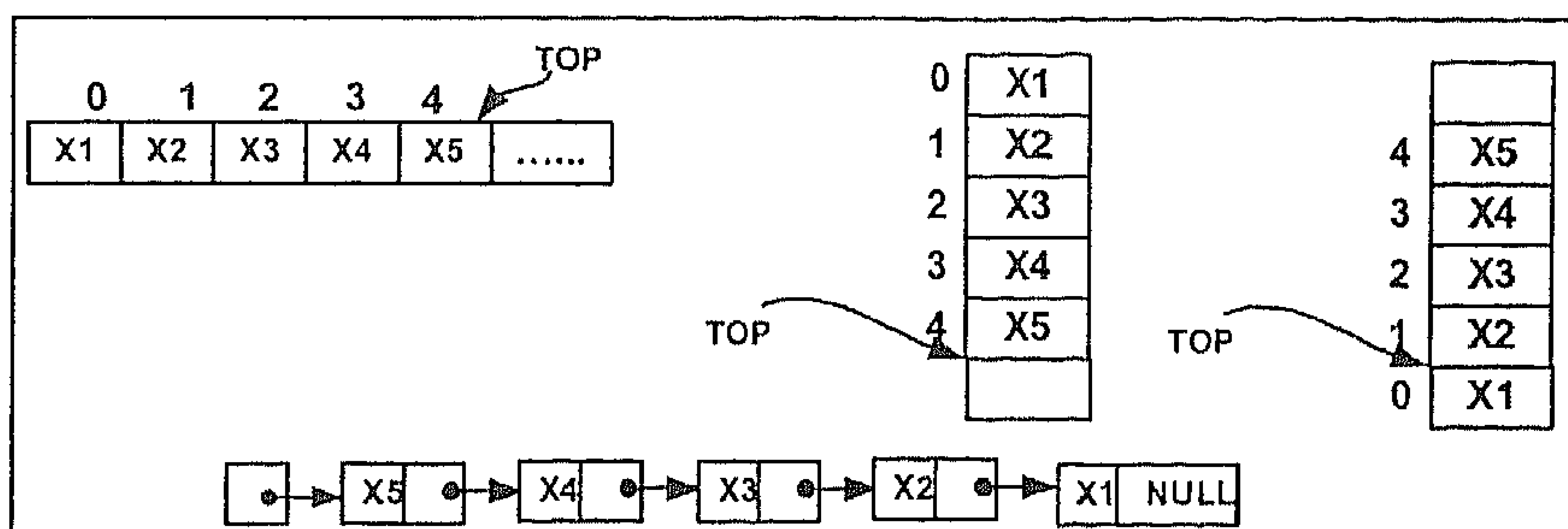
د. إن طبيعة عمل المكدس لا تستلزم وجود أكثر من مؤشر واحد للإشارة إلى مقدمة المكدس هو ما نشير إليه بمصطلح القمة (TOP).

وهناك أكثر من أسلوب للتعبير عن المكدسات من الناحية المنطقية. وقد جرت العادة على اختيار أكثر الطرق ملائمة للواقع. والشكل (1) يعطي أربعة أساليب مختلفة للتعبير عن المكدسات، إلا أن أولها هو أكثر شيوعاً.

وكما هو واضح من هذه الأساليب الأربعة في تمثيل المكدسات من الناحية المنطقية، فإن تمثيل المكدسات في الذاكرة يمكن أن يستخدم أحد أسلوبين، شأنها في ذلك شأن الطوابير:

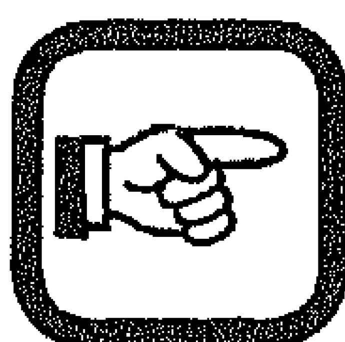
الأول هو التمثيل التتابعي باستخدام المصفوفات الأحادية، والآخر هو التمثيل المتصل

باستخدام القوائم المتصلة والمؤشرات. وبالإضافة إلى ذلك فإن من الممكن أيضاً استخدام المصفوفات المتوازية أو المصفوفات المركبة والدالات النسبية في لغات البرمجة التي لا توفر إمكانات برمجية لاستخدام المؤشرات وذلك على النحو الذي بيناه عند الحديث عن القوائم المتصلة. ولا نفضل استخدام هذا الأسلوب نظراً لأنه يخفي في ثناياه الصورة الحقيقية للمكدس.



الشكل (1): التصور المنطقي للمكدسات

ولكي نبين الفرق بين أسلوب استخدام المصفوفات الأحادية واستخدام القوائم المتصلة والمؤشرات، دعنا نورد المثال التالي:

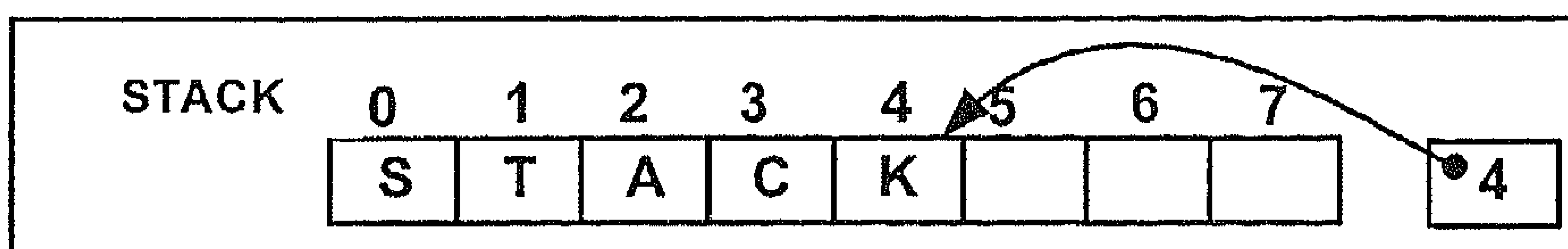


مثال (1)

افترض أن لدينا مكدساً مكوناً من خمسة قيم على النحو الموضح في الشكل (1) وهذه القيم هي على التوالي (من اليمين إلى اليسار): (K, C, A, T, S). وافترض لغايات هذا المثال أننا نرغب في تخزين هذه القيم في الذاكرة باستخدام المصفوفات الأحادية، فإن ذلك يمكن أن يعبر عنه في لغة سي++ بالصيغة البرمجية التالية:

```
typedef char    stackType[N];
  stackType stack;
  int top;
```

وبناءً على هذه التعريفات فإن المكدس سيتخذ الشكل (2) التالي:



الشكل (2): تمثيل المكدس باستخدام المصفوفات الأحادية

ولكن يفضل وضع المؤشر Top ومصفوفة البيانات في سجل واحد كما يلي:

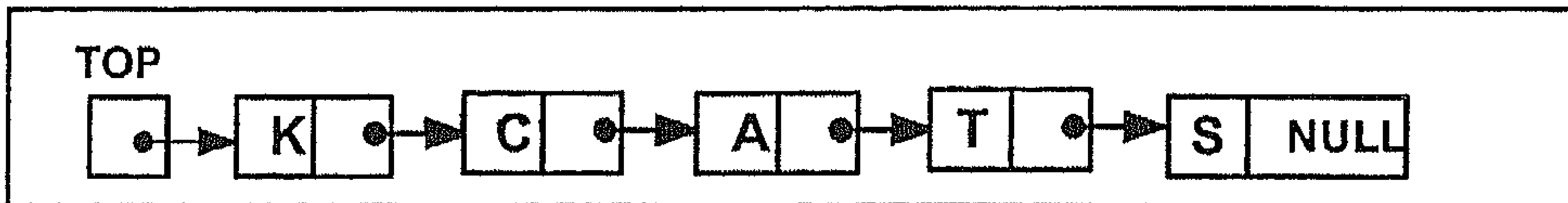
```
struct stack
   {char data[N];
   int  top;
   };
```

في مثالنا النمط البياني للعناصر من نوع Char وهي تعتمد على طبيعة التطبيق.

ومن الممكن، من ناحية أخرى تمثيل هذا المكدس في الذاكرة باستخدام القوائم المتصلة والمؤشرات. وفي هذه الحالة يمكن أن نعبر عنه في لغة سي++ بالصيغة البرمجية التالية:

```
class Ptrstack
  {public:
     char value;
     stackPtr* next;
} top;
```

وبناء على هذه التعريفات فإن المكدس سيتخذ الشكل (3) التالي:



الشكل (3): تمثيل المكدس باستخدام القوائم المتصلة والمؤشرات

وكما تلاحظ، عزيزي الدارس، فإن الفرق بين الأسلوبين من حيث التخزين لا يؤدي إلى اختلاف في أسلوب العمل. فالمتغير (top) يشير في كليهما إلى قمّة المكدس. وحذف أحد العناصر من المكدس سيؤدي إلى إجراء تعديل عليه بحيث يصبح في وضع يشير فيه إلى العنصر التالي في المكدس. وإضافة أحد العناصر يؤدي إلى تعديل قيمة هذا المتغير بحيث يصبح في وضع يشير فيه إلى العنصر الجديد. وبينما نستخدم في الأسلوب الأول طريقة الزيادة أو النقصان بمعدل واحد للمتغير (top)، وذلك على النحو المعروف في التعامل مع المصفوفات الأحادية، فإننا في الحالة الثانية (أي باستخدام التمثيل المتصل) نقوم بتغيير قيمة العنوان المطلق للمتغير (top) ليصبح في وضع يشير فيه إلى العنصر التالي ويتم هذا بصورة غير مباشرة من جانب المبرمج وذلك على النحو المعروف في القوائم المتصلة.

وأياً كان أسلوب تمثيل المكدس في الذاكرة، فإنه بهذه الصيغة التي يتخذها يمكن أن يستخدم لقلب الترتيب الذي ترد به القيم رأساً على عقب. فبناء على ما هو وارد في الشكلين (2) و(3) يمكن أن نعكس ترتيب القيم بعد إجراء العمليات المتوالية للحذف بحيث تكون النتيجة: (kcats) بدلاً من (stack).

وهناك، في الحقيقة العديد من التطبيقات التي تستخدم فيها المكدسات، فهي تستخدم في لغات البرمجة في مجالين رئيسين: الأول هو تنظيم عملية استدعاء البرامج الفرعية، والثاني هو عملية الاستدعاء الذاتي (recursion). ففي عملية الاستدعاء الاعتيادية لا بد من وجود وسيلة للمحافظة على البيانات الخاصة بالمستدعي (caller) والمستدعى (callee) كليهما، ولتحديد نقاط الدخول والخروج والعودة. فعندما يقوم أحد البرامج باستدعاء برنامج فرعي معين، فإن الأمر يقتضي الحفاظ على الحالة التي وصل إليها البرنامج المستدعي، كما يقتضي الأمر الحفاظ على البيانات الخاصة بالبرنامج الفرعي، وتحديد نقطتي الدخول والخروج. وتستخدم بعض لغات البرمجة ما يسمى بسجل النشــاط (activation record:AR) لهذه الغاية. إذ يكون لكل برنامج عامل في لحظة معينة سجل من هذا النوع، وتوضع هذه السجلات جميعاً على مكدس وفق تسلسل خطوات الاستدعاء والتنفيذ.

أما فيما يتصل بالاستدعاء الذاتي، الذي سنفرد له وحدة خاصة في هذا المقرر، فإن الوصول إلى النتيجة النهائية لعملية من العمليات التي تقوم على فكرة الاستدعاء الذاتي يعتمد على الوصول إلى نتائج جزئية. فكل نتيجة جزئية تؤدي إلى النتيجة التالية: وهكذا حتى يتم استخلاص النتيجة النهائية. والمحافظة على مقومات هذه النتائج الجزئية من بيانات وعمليات ضمن تسلسلها الجزئي المتوالي يستلزم استخدام المكدسات لهذه الغاية.

وبالإضافة إلى هذا النوع من الاستخدام فإن المكدسات تستخدم في تطبيقات مهمة أخرى لها علاقة أيضاً بلغات البرمجة. ومن أهمها تحويل التعابير الرياضية من وضعها الاعتيادي الذي يضع رموز العمليات في أماكنها المناسبة بين المعاملات (وهو ما نشير إليه باسم الإشارة الوسطية (infix) إلى صيغة جديدة تظهر فيها رموز العمليات بعد المعاملات المتصلة بها (وهو ما نشير إليه باسم نظام الإشارة التبعية (postfix) ومن ثم إجراء التقييم اللازم على التعبير الحسابي في صيغته الجديدة. وسنفرد لهذا الموضوع قسماً خاصاً، نعالج فيه كيفية استخدام المكدسات لهذه الغاية والخوارزميات ذات الصلة بها.

وبالإضافة إلى هذا وذلك فإن للمكدسات استخدامات أخرى في بعض العمليات التي

تجري على البيانات، ومن أبرزها عمليات الفرز. فهناك ما يسمى أسلوب الفرز السريع (quicksort) القائم على فكرة تجزيء مجموعة القيم المراد فرزها إلى مجموعات جزئية يتم تنظيم عملية فرزها باستخدام المكدسات. ونظراً لأن هناك وحدة مستقلة من هذا المقرر مخصصة للحديث عن عمليات الاستقصاء والفرز، فإن المقام هنا لا يسمح بالخوض في تفاصيل هذا النوع من أنواع الفرز. وهناك بالإضافة إلى ذلك أيضاً عملية استعراض عناصر الهياكل الشجرية حيث من الممكن استخدام المكدسات لهذه الغاية.

أسئلة التقويم الذاتي (1)

املأ الفراغ في العبارات التالية:

1. تعمل المكدسات وفق مبدأ ....
2. يوفر المكدّس إمكانية الوصول المباشر إلى العناصر، إلا أن هذا الوصول محدود بـ ....
3. يطلق على المتغير الذي يشير إلى آخر عنصر في المكدس اسم ....
4. هناك طريقتان أساسيتان لتمثيل المكدسات في الذاكرة هما ....
5. إذا تساوت عمليات الحذف مع عمليات الإضافة فإن المكدس يصل إلى وضع ....
6. يطلق على عملية الإضافة إلى المكدس اسم (بالعربية أو الإنجليزية) .... أما عملية الحذف فيطلق عليها اسم (بالعربية أو الأنجليزية) ....
7. تستخدم المكدسات في خمسة مجالات رئيسة تم ذكرها خلال هذا القسم هي ....

3. العمليات الخاصة بالمكدسات

1.3 وصف مجرد للعمليات

إن أهم عمليات المكدسات هي عملية الإضافة أو الدفع (Push) وعملية الحذف أو الإطلاق (Pop). بالإضافة إلى عملية تهيئة المكدس للاستخدام لأول مرة StackCreate وعمليتا فحص المكدس للتأكد من أنه فارغ أو ممتلئ (حالة الفائض) إذ أنه من الضروري قبل محاولة حذف عنصراً من المكدس التأكد من أنه غير فارغ وأيضاً من الضروري قبل محاولة إضافة عنصراً ما إلى المكدس التأكد من أنه غير ممتلئ.

وفيما يلي، عزيزي الدارس، وصفاً مجرداً لهذه العمليات وسنناقش لاحقاً كيفية تنفيذها:

1. **إنشاء مكدس StackCreate** وتستخدم هذه العملية لتهيئة المكدس للاستخدام لأول مرة

void stackCreate(AStack& Q);

2. **فحص حالة الامتلاء أو الفائض StackFull.**

وهذه العملية تعيد لنا القيمة البولينية True إذا كان المكدس ممتلئاً (Full) تماماً أي إذا وصل المكدس إلى حالة الفائض. وتعيد لنا القيمة false إذا لم يكن المكدس ممتلئاً.

bool stackFull(AStack Q);

3. **فحص فيما إذا كان المكدس خالياً StackEmpty**

وتعيد هذه العملية القيمة True إذا كان المكدس خالياً والقيمة false إذا لم يكن خالياً

bool stackEmpty(AStack Q);

4. **عملية الإضافة Push**

وتقوم هذه العملية Push بإضافة عنصر جديد ((element إلى المكدس إذا لم يكن ممتلئاً ونوعية هذا العنصر تعتمد على طبيعة البرنامج والتطبيق المستخدم للمكدس.

void push(AStack& Q, int element);

5. **عملية الحذف Pop**

وتقوم هذه العملية بحذف عنصراً من المكدس إذا لم يكن هذا المكدس فارغاً. وتعيد لنا قيمة العنصر المحذوف من خلال المعامل (element).

void pop(AStack& Q, int& element);

2.3 تنفيذ العمليات الخاصة بالمكدسات

سنناقش الآن، عزيزي الدارس، كيفية تنفيذ العمليات آنفة الذكر وسنأخذ بعين الاعتبار طريقتي تمثيل المكدسات (المصفوفات والقوائم المتصلة).

1. إنشاء المكدس StackCreate

في حالة التمثيل التتابعي (المصفوفات) فإن كل ما تقوم به StackCreate هو إعطاء (top) القيمة الابتدائية 1- وعليه فإننا نستطيع كتابته كما يلي:

```
void AStack::stackCreate(AStack& Q)
 { Q.top= -1;
 }
```

أما في حالة التمثيل المتصل (القوائم المتصلة فإن StackCreate يعطي (top) القيمة الابتدائية (NULL) وعليه فإننا نستطيع كتابته كما يلي:

```
void PtrStack::stackCreate()
 { top=NULL;
 }
```

2. فحص حالة الفائض (الامتلاء) StackFull

في حالة التمثيل التتابعي (المصفوفات) يكون المكدس ممتلئاً تماماً إذا وصلت قيمة (top) إلى أكبر قيمة وهي حجم المصفوفة المحجوزة للمكدس ويمكن أن تعرف هذه القيمة في جزء الثوابت من البرنامج (const) كما فعلنا عندما عرفنا المكدسات ونستطيع كتابة StackFull كما يلي:

```
bool AStack::stackFull(AStack Q)
    {if(Q.top==N-1)
                return true;
                else return false;
    }
```

أما في حالة التمثيل المتصل (القوائم المتصلة) فإن المكدس يكون ممتلئاً عندما تفشل عملية (new) في تخصيص ذاكرة للعنصر الجديد عندها يعطي الحاسوب رسالة خطأ ويتوقف (للأسف) تنفيذ البرنامج.

3. فحص فيما إذا كان المكدس خالياً StackEmpty في حالة التمثيل التتابعي يكون المكدس خالياً إذا كانت قيمة (top) صفرا وفي حالة التمثيل المتصل يكون

المكدس خالياً إذا كانت قيمة المؤشر (top) القيمة (NULL) وعليه فإننا نستطيع كتابة StackEmpty كما يلي:

```
bool AStack::stackEmpty(AStack Q)
    {
     if(Q.top==0)
              return true;
              else return false; }
```

وفي حالة التمثيل المتصل كما يلي:

```
bool PtrStack::stackEmpty()
  {if(top==NULL)
     return true;
    else return false; }
```

عملية الإضافة (Push)

تفترض عملية الإضافة الوجود المسبق للتركيب البياني المعني، وهو المكدس في هذا المقام. وقد يكون التركيب البياني خالياً من القيم، وبذلك يصبح العنصر المضاف هو الوحيد في المكدس. وعليه، فإن نمو المكدس يبدأ مع إضافة أول عنصر إليه، ويزداد هذا النمو مع كل إضافة جديدة إليه. ويعبر عن هذا النمو بإجراء التعديل المناسب على المتغير (top) الذي يشير إلى آخر عنصر أضيف إلى المكدس. وعندما يكون المكدس خالياً من القيم فإن قيمته تكون صفراً أو (NULL).

وعند تمثيل المكدسات باستخدام المصفوفات الأحادية، فإننا نحجز عدداً من المواضع يكفي لعدد العناصر التي يتكون منها المكدس. إلا أن المشكلة التي نصادفها في هذا الشأن هي أن حجم المكدس المطلوب قد لا يكون غير معروفٍ مسبقاً والقيم تضاف إليه على نحو غير متوقع. ومن هنا نقول بأن عدد المواضع المحجوزة هو عدد تقريبي. فقد ينمو حجم المكدس إلى حد تصبح فيه هذه الأماكن غير كافية للمزيد من الإضافة، وقد يقل عدد العناصر عن هذا الحد الأعلى المقرر. وتتم عملية الإضافة إلى المكدّس بأسلوب بسيط وخطوات محدودة.

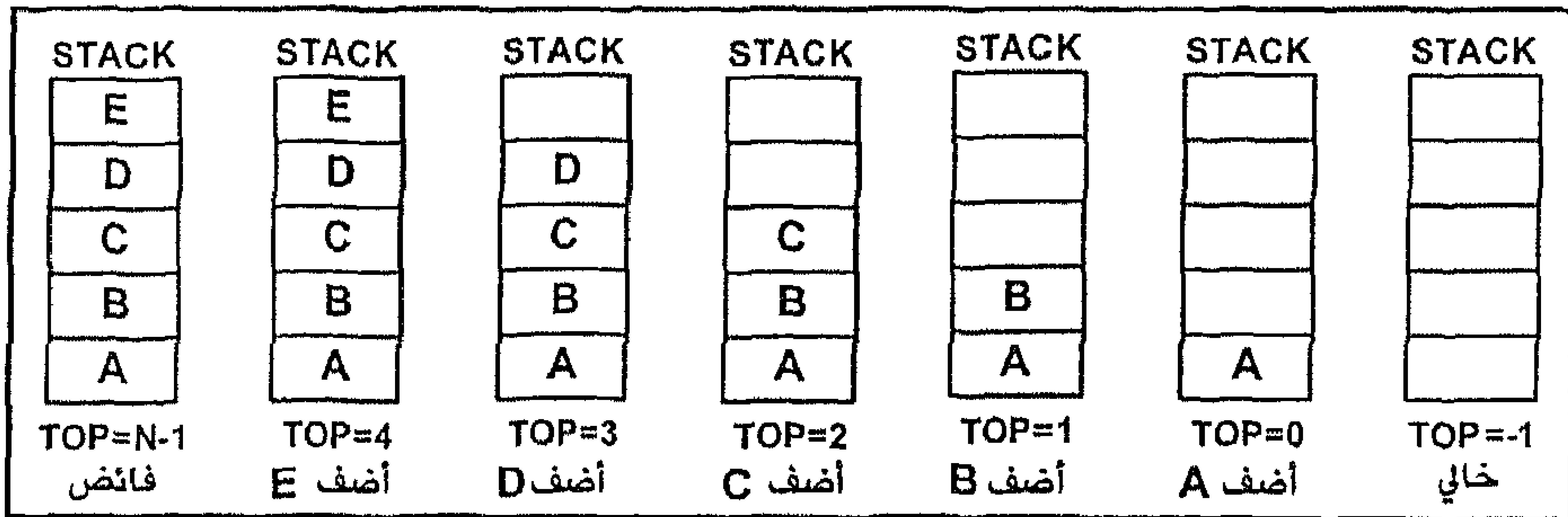
وفيما يلي نستعرض تفاصيل القيام بعملية الإضافة إلى المكدس الممثل على هيئة مصفوفة أحادية، مع التذكير بأن الخوارزميات التالية مبنية على التعريفات البرمجية التي وردت في القسم السابق.

الخوارزمية (1)

```
void AStack::push(AStack& s, int element)
    {//pushing an element on array-based stack
     if (s.top==N)
   cout<<"the stack is full .. overflow';
```

```
else
 {s.top ++;
  s.data[s.top]=element;
 }}
```

وكما تلاحظ من هذه الخوارزمية، فإن لدينا وضعين مختلفين: الأول هو عندما تكون الإضافة غير جائزة لعدم توفر الحيز، والثاني عند تنفيذ عملية الإضافة في غير ذلك. فلو كان لدينا مكدس يتسع لخمسة عناصر، فإن إضافة العناصر إليه حتى يصل إلى وضع الفائض (الامتلاء) يمكن توضيحه على النحو الوارد في الشكل (4).



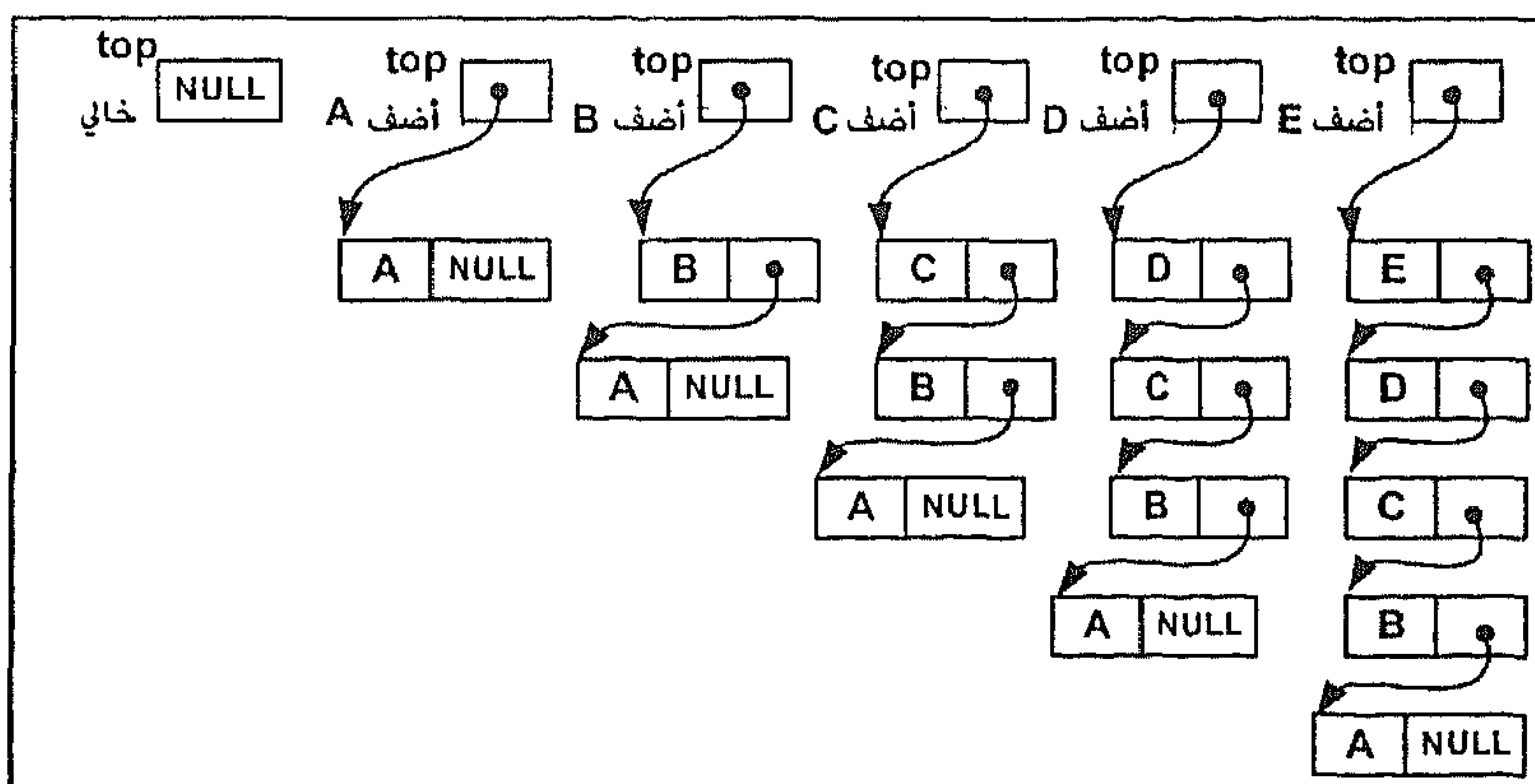
الشكل (4): انتقال المكدس (من اليمين إلى اليسار) من وضع الخالي إلى وضع الفائض من خلال الإضافة.

هذا بالنسبة لعملية الإضافة عند تمثيل المكدسات تتابعياً. أما في حالة التمثيل المتصل واستخدام المؤشرات فإن الأمر لا يحتاج إلى تحديد حد أعلى لعدد العناصر. فعملية الحجز تتم بطريقة ديناميكية وفقاً للحاجة. وبينما يتم التعبير عن الإضافة بزيادة قيمة المتغير (top) على النحو المذكور في الخوارزمية (1)، فإن التعديل الذي يجري على قيمة هذا المتغير غير منظور للمبرمج. إلا أن على المبرمج أن يحدد اتجاه هذا المؤشر بعد كل عملية إضافة. وفيما يلي نستعرض خطوات القيام بعملية الإضافة إلى المكدس الممثل على هيئة قائمة متصلة.

الخوارزمية (2)

```
void PtrStack::push(int element)
 {//pushing an element onto a stack represented as a linked list
  SPtr *ptr;
  ptr= new SPtr; // create a new node
      ptr->value= element;
  ptr->next=top;
  top=ptr;
 }
```

وكما تلاحظ، فإن هذه الخوارزمية لا تشتمل على ما اشتملت عليه الخوارزمية (1) من اختبار لحالة الفائض في المكدس، إذ ليس هناك فائض في هذا النوع من التمثيل. ثم لاحظ أيضاً كيفية ربط العنصر الجديد بالعناصر السابقة، وتحويل اتجاه المؤشر (top) في الجملة الرابعة ليشير إلى العنصر الجديد. ولو حاولنا إعادة إجراء عملية الإضافة التي تمت في الشكل (4) أعلاه، لوجدنا أنها ستتخذ السياق التالي الموضح في الشكل (5).



الشكل (5): الإضافة إلى مكدس ممثل على شكل قائمة متصلة

عملية الحذف (Pop)

إن مدى نمو المكدس يتأثر بمقدار الحذف الذي يتم عليه. وكما ذكرنا من قبل، إذا تساوى عدد مرات الحذف مع عدد مرات الإضافة فإن المكدس سيصل إلى وضع يصبح فيه خالياً من القيم. ومعنى ذلك أن المكدس يتقلص مع كل عملية حذف، وبالتالي فإن فرصة العناصر الموجودة على المكدس في الاقتراب من السطح تتزايد أيضاً.

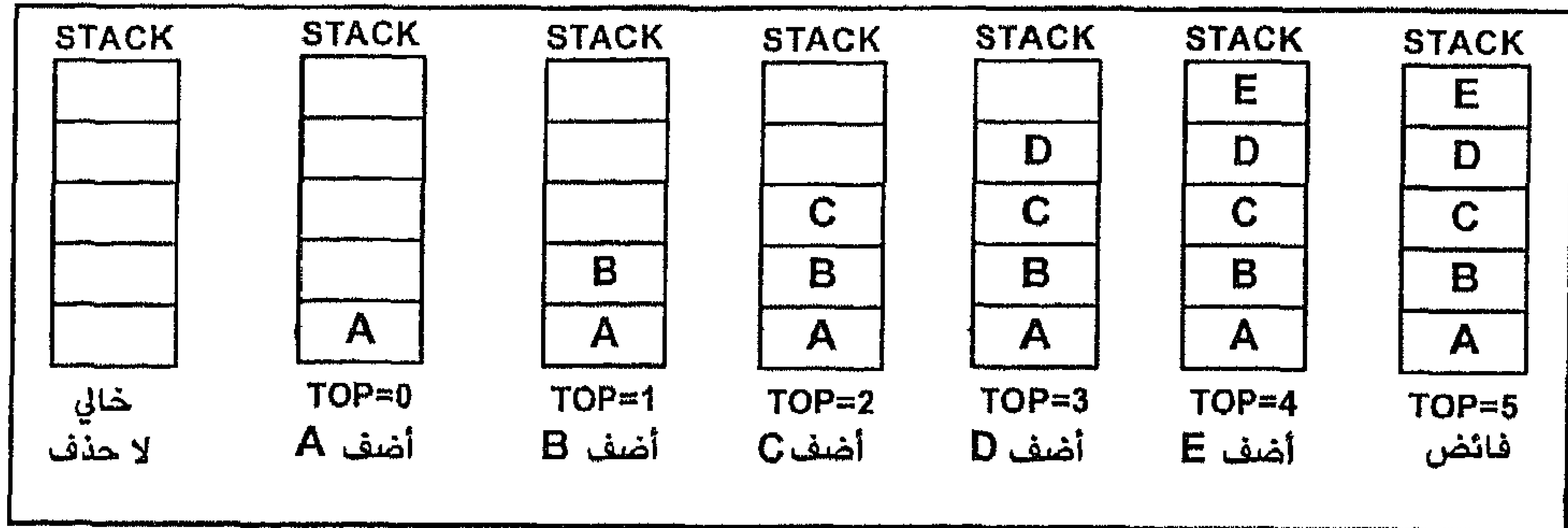
وعند تمثيل المكدسات باستخدام المصفوفات الأحادية، فإن عملية الحذف تعني إخلاء أحد المواضع وبالتالي إتاحة الفرصة لإضافة جديدة. وكما هو الحال بالنسبة لعملية الإضافة، فإن عملية الحذف تتم بأسلوب بسيط. وفيما يلي نستعرض خطوات القيام بعملية الحذف من المكدس الممثل تتابعياً على هيئة مصفوفة أحادية.

الخوارزمية (3):

```
void AStack::pop(AStack& s, int& element)
   {//popping an element from an array-based stack
   if(s.top== -1)
```

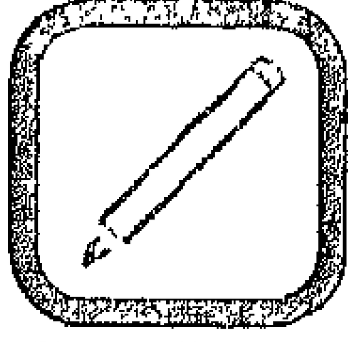
```
cout<<"stack is empty .. underflow";
else
  {element=s.data[s.top];
   s.top--;
  }
}
```

وكما لاحظنا في عمليّة الإضافة، فإن لدينا وضعين رئيسين: الأول عندما يكون الحذف غير جائز لعدم وجود قيم في المكدس، والثاني عند تنفيذ عملية الحذف في الأحوال العادية. وبناءً عليه، لو كان لدينا المكدس الوارد في الشكل (4) بصورته التي انتهى إليها ونرغب في إجراء عملية الحذف على القيم التي يضمها لوصلنا إلى الوضع الابتدائي لهذا المكدس وهو خلوّه من القيم، وذلك على النحو الوارد في الشكل (6):



الشكل (6): انتقال المكدس من وضع الفائض إلى وضع الخالي بعد الحذف

على أنه ينبغي أن لا يفهم من كلامنا هذا ومما ورد في الشكلين (4)، (6) على أننا ننتظر حتى يكتمل المكدس ويصل إلى وضع الفائض حتى نبدأ بعملية الحذف. فكما أن الإضافة تبدأ في أي وقت، لأن الحالات تأتينا (في حقيقة الأمر) بطريقة عشوائية (random) غير متوقعة، فكذا الأمر بالنسبة لعملية الحذف. ومعنى ذلك أن عمليتي الحذف والإضافة قد تتداخلان. ويعتمد هذا على نوع التطبيق التي يستخدم فيها المكدس. ولكي تتحقق من هذا الوضع بنفسك، أجب،عزيزي الدارس، على التدريب التالي.



تدريب (1)

افترض أن لدينا أحد البرامج المكتوبة بلغة سي++، على سبيل المثال، وأن هذا البرنامج يتضمن سلسلة من عمليات الاستدعاء الاعتيادية على النحو المذكور في الجدول التالي تباعاً:

البرنامج	يستدعي البرنامج الفرعي
A	B
B	C
B	D
C	E
D	F
E	G

والمطلوب منك هو أن تبين بالرسم تسلسل عمليات الإضافة والحذف التي ستتم على سجلات النشاط (ARs) الخاصة بهذه البرامج والموضوعة على مكدس لهذه الغاية. تذكر بأن أول سجل يوضع على المكدس هو ذلك السجل الخاص بالبرنامج A، وبعد أن تنتهي من الإجابة قارن إجابتك بالإجابة المعطاة في نهاية الوحدة.

والملاحظة الأساسية التي ينبغي أن تدركها في هذا الشأن، هي أن إنجاز بعض البرامج المذكورة أعلاه (في التدريب) يعتمد على استدعاء وتنفيذ بعض البرامج الفرعية الأخرى. ومعنى ذلك أن السجل الخاص بأحد البرامج ينبغي أن يوضع على المكدس حتى ننتهي من تنفيذ البرامج التي تم استدعاؤها. ومن هذا المنطلق وضعنا على المكدس أولاً السجل الخاص بالبرنامج "A"، ثم السجل الخاص بالبرنامج "B" ثم السجل الخاص بالبرنامج "C"... وهكذا. ولكن، كما تلاحظ، هناك بعض البرامج الفرعية لا تستدعي برامج فرعية أخرى مثل "F" و"G". ومعنى ذلك أنها تمثل نقطة نهاية بالنسبة لسلسلة معينة من الإستدعاءات. وهذا يعني أنه بمجرد انتهاء تنفيذ البرنامج الفرعي "G" تبدأ علمية الحذف الأولى متلوة بعملية الحذف الثانية حتى نصل إلى السجل الخاص بالبرنامج الفرعي "B" حيث يتم استدعاء "D" والذي بدوره أيضاً يقوم باستدعاء "F". وبعد تنفيذ هذا الأخير تبدأ عملية الحذف من جديد حتى ننتهي من "A".

هذه هي عمليّة الحذف من المكدسات الممثلة باستخدام المصفوفات الأحادية. أما عمليّة الحذف من المكدسات الممثلة على شكل قوائم متصلة فإنها تتم بالدرجة نفسها من المرونة والبساطة. وفيما يلي نستعرض خطوات عملية الحذف من خلال الخوارزمية (4).

الخوارزمية (4):

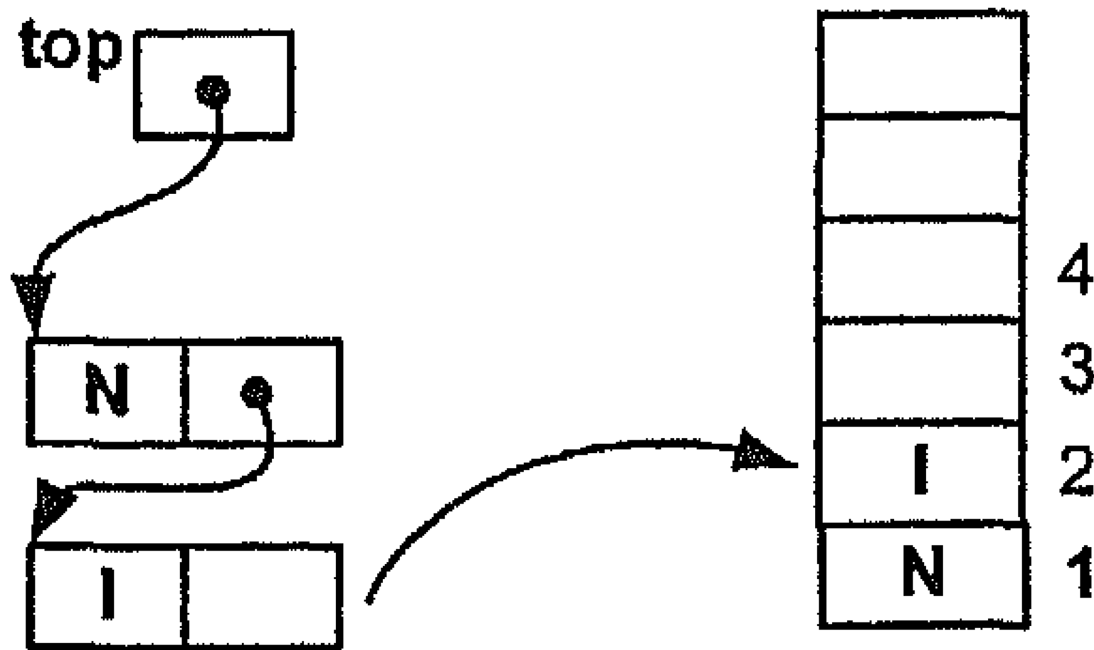
```
void PtrStack::pop(int element)
 {// popping an element from a stack implemented as a linked list
  SPtr *ptr;
  if (top==NULL)
   cout<<"stack is empty.. underflow";
   else
    {element=top->value;
     ptr=top;
     top=top->next; // update top
     delete(ptr); // delete from memory
     }
 }
```

وكما تلاحظ، فإن هذه الخوارزمية لا تختلف عن الخوارزمية (3) السابقة إلا في أسلوب كتابة الجمل. أما أوضاع الحذف فهي متماثلة. فكما هو الحال في التمثيل التتابعي، لدينا هنا وضعان متميزان: أولهما عندما يكون المكدس خالياً، وثانيهما هو الوضع الاعتيادي للحذف. ولعلك تلاحظ أيضاً بأننا استخدمنا متغيراً محلياً هو (Ptr) وذلك من أجل إجراء الحذف الفيزيائي من الذاكرة على العنصر المحذوف من المكدس باستخدام عملية (Dispose). ولو تأملنا الشكل (5) مرة أخرى لوجدنا أن عملية الحذف ستتخذ اتجاهاً معاكساً (من اليمين إلى اليسار). حيث سيتم حذف العنصر المتمثل بالقيمة "E" أولاً، متلوة بالعنصر المتمثل بالقيمة "D"... وهكذا حتى يصبح المكدس خالياً من العناصر. وبالطبع فإن الملاحظة التي أوردناها في سياق الحديث عن التدريب (1) تنطبق هنا أيضاً. ونعود فنؤكد مرة أخرى بأن عمليات الإضافة والحذف يمكن أن تتداخل، فسياق حدوثها يتوقف على منطق المشكلة المراد حلها، كما رأينا في التدريب السابق.



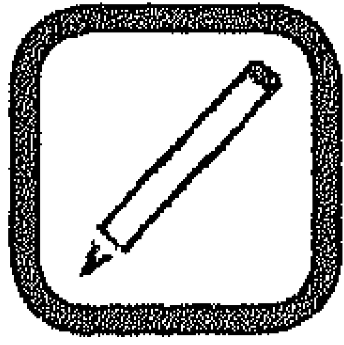
تدريب (2)

تأمل الشكل (7) التالي لمكدس مكوّن من عنصرين، ثم بيّن بالرسم وباستخدام خوارزميات الإضافة والحذف، التي ورد ذكرها فيما سبق، الوضع الذي سيؤول إليه المكدس بعد كل عملية من عمليات الإضافة والحذف الواردة في الجدول المصاحب.



العملية	العنصر
إضافة	x
حذف	-
حذف	-
حذف	y
حذف	-
حذف	-

الشكل (7): الإضافة والحذف في المكدسات



تدريب (3)

مستعيناً بخوارزميتي الإضافة والحذف الخاصتين بالمكدسات الممثلة باستخدام القوائم المتصلة والمؤشرات، اكتب برنامجاً بسيطاً يقوم بطباعة الحروف الهجائية الإنجليزية بشكل معكوس، أي ابتداء من حرف "Z" وحتى حرف "A" وذلك على النحو التالي (من اليسار إلى اليمين):

(Z, Y, X, W ..., C, B, A)

تدريب (4)

افترض أن لدينا مكدساً مكوناً من أربعة قيم على النحو المبين في الشكل (7)، وأن لدينا سلسلة العمليات التالية التي نريد تنفيذها تباعاً، فما هي إجابتك على الأسئلة المذكورة أدناه:

```
element='N';
top--;
push (stack, top, element);
push (stack, top, 'r');
push (stack, top, 'm');
push (stack, top, 'd');
push (stack, top, 's');
push (stack, top, 'e');
push (stack, top, 'f');
for (int i=1; i<=n;i++)
pop(stack, top);
```

أ. ما قيمة top المبدئية وفقاً للشكل (7)؟

ب. ما الحد الأعلى (n) المسموح به لعدد عناصر المكدس وفقاً للشكل نفسه؟

ج. ما الحد الأدنى الذي يمكن أن يصل إليه المكدس؟

د. ما نتيجة تنفيذ سلسلة الجمل حتى بداية التركيب الدوراني for (بين ذلك بالرسم)؟

هـ. ما نتيجة تنفيذ التركيب الدوراني المذكور (بين ذلك بالرسم)؟

و. ما قيمة المتغير top بعد تنفيذ جملة الدوران for للمرة الأخيرة؟

4. تطبيقات على استخدام المكدسات

سنعرض في هذا القسم، عزيزي الدارس، بعض التطبيقات المهمة التي تستخدم المكدسات، ونخصص الوحدة التالية (الثامنة) لبحث موضوع الاستدعاء الذاتي الذي هو أيضاً من أهم تطبيقات المكدسات.

1.4 إيجاد قيمة تعبير رياضي (حسابي) باستخدام المكدسات

لقد تعلمت، عزيزي الدارس، في الرياضيات بأن هناك أولويات محددة للقيام بالعمليات الحسابية. فعملية الضرب، مثلاً، ينبغي أن تتم قبل الجمع. فلو كان لدينا التعبير الحسابي التالي: "A+B*C" فإننا نقوم أولاً بضرب قيمة "B" بقيمة "C"، ومن ثم نجمع ناتج الضرب إلى قيمة "A". ولو لم يكن لدينا مثل هذا النظام وعوضنا عن التعبير الحسابي المذكور بالقيم الخاصة بالمعاملات على النحو التالي: " 5+4*10 " فإن من الممكن أن نقوم بتقييم هذا التعبير بطريقتين مختلفتين، ونحصل بالتالي على نتيجتين مختلفتين. فمن الممكن أن نجمع أولاً ثم نضرب وتكون النتيجة بذلك هي "90". ومن الممكن، من زاوية أخرى، أن نضرب أولاً ومن ثم نجمع، وتكون النتيجة بذلك هي "45". وشتان بين النتيجتين، كما ترى. وكمثال آخر، تأمل التعبير الأسي التالي. فهذا التعبير يمكن أن يعني شيئين مختلفين. فقد يؤخذ على أنه يعني أو أنه يعني . وبينما يعطي المعنى الأول النتيجة "64" فإن المعنى الآخر يقدم لنا النتيجة "512".

ومن أجل ذلك تلجأ لغات البرمجة إلى تبنّي سياسة محددة في القيام بتقييم التعابير الرياضية، وإلى الإفادة من المكدسات لهذه الغاية. ولكن قبل أن نبين كيف يتم ذلك فإن من المهم ملاحظة أن التعابير الرياضية ليست سهلة دائماً، كما قد يتبادر إلى الذهن. وأحد أسباب التعقيد ينبع من استخدام الأقواس في التعابير الرياضية، وذلك لضمان أن بعض العمليات ينبغي أن تتم بسياق محدد يقرره منطق حل المسألة المعنية. ومن أجل تسهيل عملية التقييم، فإن المترجم يلجأ إلى تحويل التعبير الرياضي من صياغته المعتادة القائمة على استخدام نظام الرموز الوسطية (infix) إلى نظام الرموز التبعية (postfix) وهي عملية سبق أن نوهنا إليها من قبل عند الحديث عن استخدامات المكدسات.

وبينما تقوم فكرة نظام الرموز الوسطية على وجود الرمز المعبر عن أحد العمليات الحسابية بين المعاملين اللذين ينتميان إليه (مثال ذلك: A+B) فإن نظام الرموز التبعية يضع رمز العملية الحسابية بعد المعاملين المقصودين بالعملية (مثال ذلك: AB+). وبعد

التحويل من النظام الأول إلى النظام الثاني فإن التعبير الرياضي يتخلص من الأقواس، وتصبح عملية تقييم التعبير سهلة التنفيذ. وبالإضافة إلى هذين النظامين، فإن هناك نظاماً ثالثاً تقوم فكرته على وضع رمز العملية الرياضية قبل المعاملين اللذين ينتميان إليه (مثال ذلك: AB+). وبذلك فإن الرمز يظهر كما لو كان اسما لاقترانٍ مكتبي مقنن، وهذا هو بالفعل ما نجده في لغة LISP (مثال ذلك: plus AB). وبينما يطلق على هذا النظام اسم الترميز البولندي (Polish notation)، نسبة إلى عالم رياضي بولندي، فإن النظام التبعي يشار إليه بالنظام البولندي المعكوس (Reverse Polish notation). والجدول (1) يقدم أمثلة للتعابير الرياضية وفق الأنظمة الثلاثة المذكورة.

جدول (1): الأنظمة المعتمدة لوضع رموز العمليات في التعابير الرياضية

[illegible]	التبعي (postfix)	الوسطي (infix)
/AB	AB/	A/B
-+ABC	AB+C-	A+B-C
/ sin XX	X sin X /	sin (X) / X
*A+BC	ABC+*	A (B+C)
+A/BC	ABC/+	A+B/C
/+ABC	AB+C/	(A+B)/C

2.4 التحويل من النظام الوسطي إلى النظام التبعي (converting infix to postfix)

قبل أن نبين لك، عزيزي الدارس، كيف تقوم بعملية التحويل، فإننا بحاجة إلى اعتماد نظام معين للأولويات. وعلى الرغم من أن لغة سي++ التي نستعملها في كتابة خوارزميات هذا المقرر، تختلف عن الكثير من اللغات الأخرى في هذه المسألة، وفي طبيعة الرموز المستعملة للتعبير عن العمليات، إلا أننا سنحاول أن نضع الأولويات التالية، وهي كما يلي تباعاً:

^ الأس (exponentiation)

*، / الضرب والقسمة (multiplication and division)

+، - الجمع والطرح (addition and subtraction)

وفي حالة تساوي الأولوية، فإن القاعدة المستعملة هي التزام السياق الذي وردت فيه من اليسار إلى اليمين، وذلك ما لم تكن هناك أقواس.

أما خطوات التحويل، فإنها تسير وفق الخطوات التالية:

أولاً: استعرض التعبير الرياضي من اليسار إلى اليمين، رمزاً بعد رمز.

ثانياً: افعل ما يلي بالنسبة لكل رمز تصادفه في التعبير:

1. إذا كان الرمز يمثل أحد المعاملات (متغيراً كان أو قيمة ثابتة)، يضاف هذا الرمز إلى الصيغة الجديدة للتعبير الرياضي (التعبير الجديد).

2. إذا كان الرمز يمثل إحدى العمليات (الضرب، القسمة،... الخ)، فافعل ما يلي:

أ. إذا لم يكن المكدس (مكدس العمليات) خالياً، فأطلق (احذف: Pop) ما على المكدس من رموز تمثل العمليات وأضفها إلى التعبير الجديد حتى تصادف أحد الأوضاع التالية:

- المكدس أصبح خالياً.
- الرمز الموجود على المكدس يحمل أولوية أقل من الرمز الذي تمت قراءته.
- هناك قوس هلالي يتجه إلى اليمين ")".

ب. سواء كان المكدس خالياً أو لم يكن كذلك، ادفع بالرمز الجديد الذي تمت قراءته إلى المكدس (مكدس العمليات).

3. إذا كان الرمز يمثل قوساً متجهاً إلى اليمين ")"، أضف هذا الرمز إلى المكدس.

4. إذا كان الرمز يمثل قوساً هلالياً يتجه إلى اليسار "("، افعل ما يلي:

أ. أضف إلى التعبير بصيغته الجديدة (التعبير الجديد) جميع رموز العمليات الموجودة على سطح المكدس واحداً بعد آخر حتى تصل إلى قوس هلالي يتجه إلى اليمين")".

ب. استبعد هذا القوس من المكدس، واستبعد معه أيضاً القوس الهلالي الآخر الذي نحن بصدده.

ثالثاً: هل انتهت عملية استعراض وقراءة التعبير الأصلي؟

أ. إذا كانت الإجابة بالإثبات، فأطلق ما تبقى على المكدس من رموز العمليات وأضف هذه الرموز إلى التعبير الجديد. وبذلك تكون عملية التحويل قد تمت.

ب. إذا كانت الإجابة بالنفي، نعود مرة أخرى إلى الخطوة الأولى لقراءة الرمز التالي من التعبير الأصلي (infix)، ويتم تكرار الخطوات مرة أخرى.

ولكي نوضح كيف تجري هذه الخطوات، دعنا نقوم بتحويل التعابير الأربعة المبينة في الجدول (2) التالي:

جدول (2)

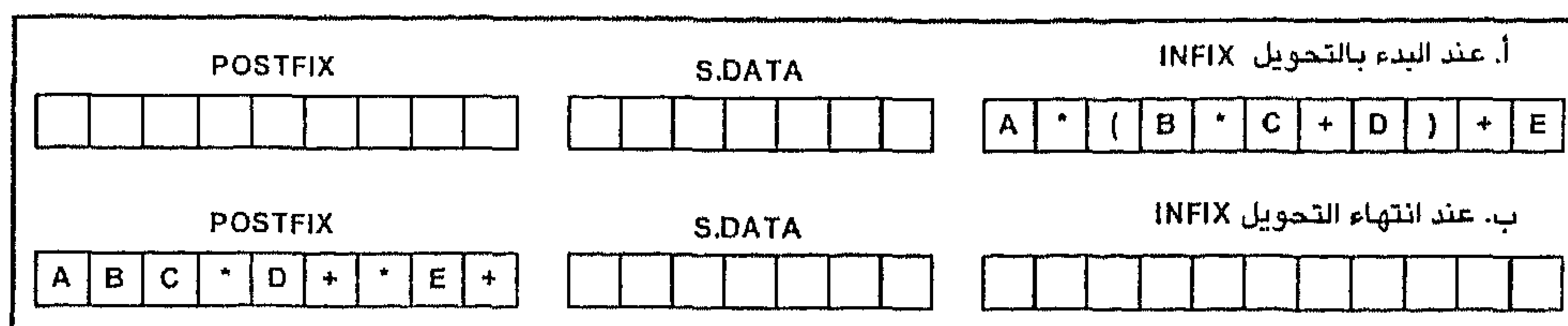
التعبير الأصلي (المراد تحويله) (prefix)	مكدس العمليات	التعبير الجديد (postfix)
A+B *C [1]	خال من القيم	خال من القيم
+B *C	خال	A
B *C	+	A
*C	+	AB
C	+*	AB
خال	*+	ABC
خال	+	ABC *
خال	خال	ABC *+
التعبير الأصلي (المراد تحويله) (prefix)	**مكدس العمليات**	**التعبير الجديد (postfix)**
A *B+C [2]	خال	خالي
*B+C	خال	A
B+C	*	A
+C	*	AB
C	+	AB *
خال	+	AB *C
خال	خال	AB *C+
التعبير الأصلي (المراد تحويله) (prefix)	**مكدس العمليات**	**التعبير الجديد (postfix)**
(A+B) *C [3]	خال	خال
A+B) *C	)	خال
+B) *C	(	A
B) *C	(+	A
) *C	(+	AB
*C	خال	AB+
C	*	AB+
خال	*	AB+C

خال	خال	AB+C *

التعبير الأصلي (المراد تحويله) (prefix)	مكدس العمليات	التعبير الجديد (postfix)
A *(B*C+D)+E [4]	خال	خالي
*(B *C+D)+E	خال	A
(B *C+D)+E	*	A
B *C+D)+E	*(	A
*C+D)+E	*(	AB
C+D)+E	*(*	AB
+D)+E	*(*	ABC
D)+E	*(+	ABC *
)+E	*(+	ABC *D
+E	*	ABC *D+
E	+	ABC *D+ *
خال	خال	ABC *D+ *E
خال	خال	ABC *D+ *E+

وكما يتضح من هذه الأمثلة، فإن عملية التحويل تستدعي وجود ثلاثة تراكيب بيانية: الأول لتخزين التعبير الرياضي (الحسابي) بصيغته الأصلية (infix) والثاني لتخزين رموز العمليات بصفة مؤقتة أثناء مرحلة التحويل، والثالث لتخزين التعبير بصيغته الجديدة (postfix). ولو تأملنا هذه الأمثلة مرة أخرى لوجدنا أن مكدس العمليات يبدأ خالياً عند البدء بعملية التحويل وينتهي خالياً. أما التركيب البياني الخاص بالتعبير الأصلي، فإنه يبدأ وبداخله جميع رموز التعبير الحسابي إلا أنه ينتهي خالياً، وذلك بعكس التركيب البياني الخاص بالتعبير الجديد الذي يبدأ خالياً وينتهي وبداخله جميع رموز التعبير باستثناء الأقواس. والشكل (8) يوضح وضع هذه التراكيب البيانية الثلاثة عند البدء بعملية التحويل وعند انتهائها مستخدمين في ذلك التعبير الرابع المذكور أعلاه.

وفيما يلي سنحاول ترجمة الخطوات المذكورة سابقاً في تحويل التعبير الوسطي (التعبير الأصلي) (infix) إلى تعبير تبعي (التعبير الجديد) (postfix) إلى خوارزمية تعرض التفاصيل الدقيقة للقيام بهذه العملية. ولكن قبل أن نستعرض خطوات هذه الخوارزمية، دعنا، عزيزي الدارس، نقوم أولاً بتعريف التراكيب البيانية والمتغيرات التي نحتاجها لهذه الغاية.



الشكل (8): التراكيب البيانية المستخدمة في عملية التحويل من infix إلى postfix

```
    const    n =20;
typedef char          charbuff[20];
    int m, p ; // last element in infix and postfix
    charbuff infix,  postfix;
typedef char StDataType;

 class AStack {
  private:
   char data[n];
   int top;
     public:
        .........
   };
```

الخوارزمية (5):

```
void  AStack::convert(charbuff infix ,charbuff& postfix,
              int  m, int& p)
  { // converting an infix expression into a postfix form
   AStack s;
   int  j;
   StDataType  element;
    stackCreate(s);
   p=0;
   for(j=0;j<n;j++)
    if(infix[j]=='^'||infix[j]=='*'||
      infix[j]=='/'||infix[j]=='+'||
      infix[j]=='-')
     {if (!stackEmpty(s))
       while(priority(s.data[s.top])>=priority(infix[j])
          && (s.top !=0) && s.data[s.top]!='(')
         {pop(s, element);
         p++;
         postfix[p]=element;
         }
```

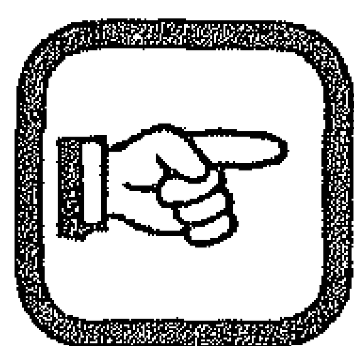
```
            push(s,infix[j]);
           }
          else
           if (infix[j]=='(')
             push(s,infix [j]);
           else
            if (infix[j]==')')
             {while (s.data[s.top]!='(')
                {pop (s,element);
                 p++;
                 postfix[p]=element;
                }// end while
              if(s.data[s.top]=='(')
               s.top=s.top-1;
             }
             else
              {p++;
               postfix[p]=infix[j];
              }

    while (!stackEmpty(s))
      { pop(s,element);
        p++;
         postfix[p]=element;
       } //end while
  }
```

ولعلك تلاحظ، عزيزي الدارس، بأن هذه الخوارزمية تقوم باستدعاء (priority) ومهمتها هي تحديد درجة أولوية العملية التي نحن بصددها خلال استعراض التعبير الحسابي. وفيما يلي تفاصيل هذا الاقتران:

```
 int AStack::priority(StDataType Operator)
   {//finding the priority of an operator
     switch (Operator)
      {case '^':return 3; break;
       case '*':return 2; break;
       case '/':return 2; break;
       case '+':return 1; break;
       case '-':return 1; break;
       case '(':return 0; break;
        }
   }
```

وكما تلاحظ، فإن لدينا أربع أولويات يعبر عنها هذا الاقتران. ويمكن زيادة هذا العدد وإجراء أي تغيير نرغب به في سلم الأولويات. ولكي نبيّن كيف تعمل هذه الخوارزمية، دعنا نورد المثال التالي:



مثال (2)

افترض أن لدينا التعبير التالي: (a+b^ c*d) * (e+f/d) ونرغب في تحويله إلى النظام التبعي (postfix).

الحل:

فالخطوة الأولى هي أن نقوم بإدخال هذا التعبير وتخزينه في المصفوفة التي رمزنا إليها بالاسم (infix). ومن ثم نقوم باستدعاء الخوارزمية على النحو التالي:

(convert (infix, postfix, m, p

وبناءً على هذا الاستدعاء، فإن لدينا أربعة متغيرات فعلية: إثنان منها معلومان لدينا ومهمتهما هي إرسال البيانات إلى الإجراء convert، وهما (infix) الذي يمثل التعبير الحسابي الذي تم إدخاله والآخر هو (m) الذي يشير إلى آخر عنصر في هذا التعبير داخل المصفوفة، وقيمته في هذه الحالة هي "17". أما المتغيران الآخران فهما للنتائج. فمهمة (postfix) هي إعادة التعبير بصيغته الجديدة، ومهمة (p) هي الإشارة إلى آخر عنصر في هذا التعبير الجديد. ومعنى ذلك أن قيمة كل منهما غير معلومة عند الاستدعاء.

أما في داخل الإجراء (convert) فتسير الأمور على النحو التالي (انظر الجدول (3)):

أولاً: يعطى المتغيران (s.top) و(p) القيمة الابتدائية صفر.

ثانياً: يبدأ استعراض الرموز المخزنة في المصفوفة (infix) واحداً بعد الآخر باستخدام التركيب الدوراني (for) وذلك على النحو التالي:

1 - j = 0

2 - ? if (infix [j] == Operator) والإجابة بالنفي.

3 - ? if (infix [j]=='(') والإجابة بالإثبات.

إذن: s.data [0] = infix [0]

4 - j = 1

5 - ? if (infix [j] == Operator) والإجابة بالنفي.

6 - ? if (infix [j]=='(') والإجابة بالنفي.

7 - ? if (infix [j]==')') والإجابة بالنفي.

8 - إذن الرمز يمثل أحد المعاملات، وبذلك فإن:

p = 0+1

postfix [0] = infix [1]

j = 2 - 9

infix [j] ==Operator ? - 10 — والإجابة بالإثبات.

إذن: أ - if (!stackempty(s)) — والإجابة بالإثبات.

ب - ؟ priority ('(') >= priority('+') — والإجابة بالنفي.

ج - s.data [1] = infix [2]

j = 3 - 11

if (infix [j]== Operator) ? - 12 — والإجابة بالنفي.

if (infix [j]== '(') ؟ - 13 — والإجابة بالنفي.

if (infix [j]== ' (') ؟ - 14 — والإجابة بالنفي.

15 - الرمز يمثل أحد المعاملات، وبذلك فإن:

p=1+1

postfix [2] = infix [4]

j = 4 - 16

if (infix [j]== Operator) ?- 17 — والإجابة بالإثبات

إذن: أ - ? if (!=stackEmpty (s)) — والإجابة بالإثبات.

ب - ? priority ('+') >= priority ('^') — والإجابة بالنفي.

ج -s.data [3] = infix [4]

j = 5- 18

if (infix [j]== Operator) ? - 19 — والإجابة بالنفي.

if (infix [j] == Operator) ? - 20 — والإجابة بالنفي.

if (infix [j] == Operator)? - 21 — والإجابة بالنفي.

22 - الرمز يمثل أحد المعاملات، وبذلك فإن:

p = 2+1

postfix [3] = infix [5]

j = 6- 23

if (infix [j] == Operator) ?- 24 — والإجابة بالإثبات.

إذن: أ - ? if (!= stackempty(s)) — والإجابة بالإثبات.

ب - priority ('^') >= priority('*') — والإجابة بالإثبات.

إذن: element ='^'

p = 3+1

postfix [4] = element

ج - s.data[3] = infix [6]

25 -7 = j

26 - if (infix [j] == Operator)? والإجابة بالنفي.

27 -؟ if (infix [j]=='(') والإجابة بالنفي.

28 - ? if (infix [j]==')') والإجابة بالنفي.

29 - الرمز يمثل أحد المعاملات، وبذلك فإن:

p = 4+1

postfix [5] = infix [7]

30 - 8 = j

31 - ? if (infix [j] ==Operator) والإجابة بالنفي.

32 -؟ if (infix [j]=='(') والإجابة بالنفي.

33 -? if (infix [j]==')') والإجابة بالإثبات.

إذن: أ- ؟ s.data[3]='(' والإجابة بالنفي.

ب- element='^'

1+p = 5

postfix [6] = element

ج- ? s.data [2]=='(' والإجابة بالنفي.

د- element='+'

1+p = 6

postfix [7] = element

هـ- ? s.data [1]=='(' والإجابة بالإثبات.

إذن: s.top = 0

34 -9 = j

35 - ? if (infix [j] == Operator) والإجابة بالإثبات.

إذن: أ- ? if (!= stackempty(s)) والإجابة بالنفي.

ب-s.data [1] = infix [9]

36 -10 = j

37 - ? if (infix [j] == Operator) والإجابة بالنفي.

38 - ? if (infix [j]=='(') والإجابة بالإثبات.

إذن: s.data [2] = infix [10]

39 - ? if (infix [j] == Operator) والإجابة بالنفي.

40 -؟ if (infix [j]=='(') والإجابة بالنفي.

41 -? if (infix [j]==')') والإجابة بالنفي.

42 -الرمز يمثل أحد المعاملات، وبذلك فإن:

1+p = 7
[postfix [8] = infix [11

j = 12- 43

if (infix [j] == Operator) ?- 44 والإجابة بالإثبات.

إذن: أ-؟ if (!= stackempty(s)) والإجابة بالإثبات.

ب- priority ('(') >= priority('+') والإجابة بالنفي.

ج- s.data[3] = infix [12]

j = 13 - 45

if (infix [j] == Operator) ?- 46 والإجابة بالنفي.

if (infix [j]=='(') ؟- 47 والإجابة بالنفي.

if (infix [j]==')') ?- 48 والإجابة بالنفي.

49 -الرمز يمثل أحد المعاملات، وبذلك فإن:

p = 8+1

postfix [9] = infix [13]

j = 14- 50

if (infix [j] == Operator) ? - 51 والإجابة بالإثبات.

إذن: أ- ? if (!= stackempty(s)) والإجابة بالإثبات.

ب- priority ('+') >= priority('/') والإجابة بالنفي.

ج- s.data[4] = infix [14]

j = 15- 52

if (infix [j] == Operator) ? - 53 والإجابة بالنفي.

if (infix [j]=='(') ؟- 54 والإجابة بالنفي.

if (infix [j]==')') ?- 55 والإجابة بالنفي.

56 -الرمز يمثل أحد المعاملات، وبذلك فإن:

p = 9+1

postfix [10] = infix [15]

j = 17 - 57

if (infix [j] == Operator) ? - 58 والإجابة بالنفي.

if (infix [j]=='(') ؟- 59 والإجابة بالنفي.

if (infix [j]==')') ?- 60 والإجابة بالاثبات.

إذن: أ - ؟ s.data[3]=='(' والإجابة بالنفي.

ب- element='/'

postfix [11] = element

ج- ? s.data [3]=='(' والإجابة بالنفي.

د- element='+'

p = 11+1

postfix [12] = element

هـ - ? s.data [2]=='(' والإجابة بالإثبات.

إذن: s.top =1

61 - if (!stackempty (s)) والإجابة بالنفي.

إذن: '*'=element p = 12+1

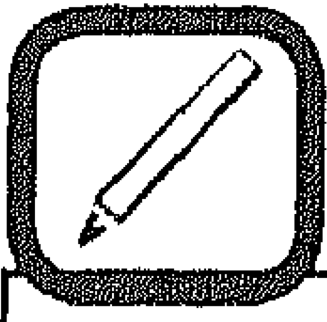
postfix [13] = element

62 -if (!stackempty (s)) والإجابة بالإثبات.

إذن: وصلنا إلى النهاية واستكملنا عملية التحويل على النحو الموضح في الجدول (3).

جدول (3): خطوات تحويل تعبير حسابي إلى النظام التبعي (postfix)

	infix	حالة الـ s.data	postfix
1	(	(	
2	a	(	a
3	+	(+	a
4	b	(+	ab
5	^	(+^	ab
6	c	(+^	abc
7	*	(+*	abc^
8	d	(+*	abc^d
9	)	empty	abc^d *+
10	*	*	abc^d *+
11	(	(*	abc^d *+
12	e	(*	abc^d *+e
13	+	+(*	abc^d *+e
14	f	+(*	abc^d *+ef
15	/	*(+/	abc^d *+ef
16	d	*(+/	abc^d *+efd
17	)	*	abc^d *+efd/+*



تدريب (5)

تأمل الخوارزمية (5) والمثال رقم (2) المبني عليها مرة أخرى، ثم قم بعملية تحويل التعبير التالي من النظام الوسطي إلى النظام التبعي (postfix):

A + (B * C - (D / E^ F) * G) * H

يفضل أن تكون إجابتك على هذا التدريب من جزئين: الأول هو تسلسل الخطوات التفصيلية على النحو الوارد في المثال (2)، والثاني هو توضيح عملية التحويل بالرسم على النحو الوارد في الجدول (3).

3.4 تقييم التعبير الحسابي (Arithmetic Expression Evaluation)

الآن وقد عرفنا كيف نقوم بتحويل تعبير حسابي من وضعه الوسطي إلى النظام التبعي، فإنه يبقى علينا أن نبين كيف نقوم بتقييم (حساب) التعبير الحسابي الناتج. والحقيقة أن العملية قد أصبحت بهذا التحويل ميسرة للغاية. إذ لم يعد لدينا أي أقواس تستدعي وضعاً استثنائياً، وأصبح بالإمكان تطبيق القاعدة العامة والمعروفة وهي السير من اليسار إلى اليمين. ونلخص فيما يلي مجموعة القواعد التي نحتاج إليها في عملية التقييم:

أولاً: استعرض التعبير الحسابي من اليسار إلى اليمين.

ثانياً: عندما تصادف رمزاً يمثل إحدى العمليات، فطبق هذه العملية على المعاملين (الرقمين) اللذين يسبقاه مباشرة.

ثالثاً: استبدل هذين المعاملين ورمز العملية بالنتيجة التي حصلنا عليها من الخطوة السابقة.

رابعاً: هل انتهى التعبير؟.

1. فإذا كانت الإجابة بالنفي أعد تكرار الخطوات السابقة.
2. إذا كانت الإجابة بالإثبات فإن نتيجة التقييم هي آخر قيمة حصلنا عليها بعد تنفيذ الخطوات السابقة.

ولكي نبين كيف نطبق هذه الخطوات الأربع السابقة، لنقم بتقييم التعابير الحسابية الثلاثة المبينة في جدول (4) التالي، علماً بأنها مذكورة أولاً بصيغة الرموز ومن ثم بالصيغة التبعية ومن ثم بتعويض القيم.

جدول (4)

التعبير الثالث	التعبير الثاني	التعبير الأول
A - B ^ C	(A + B) ⁕ C	A + B ⁕ C
A B C ^-	A B + C ⁕	A B C⁕+
922^-	723+⁕	345⁕+
9(22^)-	(72+)3⁕	3(45⁕)+
94-	93⁕	320+
النتيجة: 5	النتيجة: 27	النتيجة: 23

ولو حاولنا تتبع ما حدث بالنسبة للتعبير الأول، على سبيل المثال، لوجدنا أننا فعلنا ما يلي:

1. قمنا باستعراض التعبير الحسابي من اليسار إلى اليمين حتى وصلنا إلى علامة الضرب "⁕".
2. قمنا بتطبيق هذه العملية على أقرب معاملين من اليسار وهما "B","C".
3. قمنا باستبدال هذين المعاملين ورمز العملية بالنتيجة الجزئية التي حصلنا عليها وبذلك أصبح الوضع هكذا: " 20 3+".
4. قمنا باستكمال عملية الاستعراض، وصادفنا هذه المرة الرمز "+" الذي يمثل عملية الجمع.
5. قمنا بتطبيق هذا الرمز على أقرب معاملين من اليسار وهما، في هذه الحالة، النتيجة الجزئية وقيمة المعامل "A".
6. قمنا باستبدال الرمز والمعاملين بالنتيجة الجديدة وبذلك أصبح الوضع هكذا: "23".
7. قمنا باستكمال عملية الاستعراض، ولكن في هذه المرة لم يبق لدينا من التعبير أية رموز أخرى. ومعنى ذلك أن العملية قد انتهت والرقم الوحيد الموجود هو نتيجة عملية التقييم.

وكما يتضح من هذه الأمثلة والخطوات التي بنيت عليها، فإن المكدس هو أنسب التراكيب البيانية للقيام بمهمة تقييم التعابير الحسابية وفق النظام التبعي. وبذلك فإن من الممكن إعادة كتابة الخطوات السابقة على النحو التالي:

أولاً: استعرض التعبير الحسابي من اليسار إلى اليمين، رمزاً بعد آخر.

ثانياً: إذ كان الرمز الذي تمت قراءته يمثل أحد المعاملات، يتم تعويض المعامل بالقيمة العددية التي يمثلها، ثم توضع هذه القيمة على المكدس.

ثالثاً: إذا كان الرمز الذي تمت قراءته يمثل إحدى العمليات، يتم ما يلي:

1. إطلاق آخر قيمتين تمت إضافتهما إلى المكدس.
2. تنفيذ العملية الحسابية التي يمثلها الرمز عليهما.
3. إضافة النتيجة الجزئية التي حصلنا عليها إلى المكدس.

ثالثاً: هل انتهت عملية استعراض التعبير الحسابي؟

1. الإجابة بالإثبات، إذن: القيمة الموجودة على المكدس هي النتيجة النهائية.
2. الإجابة بالنفي، إذن: أعد تكرار الخطوات السابقة.

ولكي نبيّن كيف نطبق هذه الخطوات، دعنا نقوم بعملية تقييم التعابير الحسابية المبينة في الجدول (5) التالي، علماً بأن قيم المعاملات هي كما يلي: "" B = 5" "C = 6" ."" A = 4

جدول (5)

التعبير الحسابي	المكدس	تنفيذ العملية الحسابية
A B + C*	خالي	–
B + C *	4	–
+ C *	4,5	–
C *	9	4 + 5=9
*	9,6	–
خالي	54	9*6=54

وفي الحقيقة أننا لو أمعنا النظر في أسلوب استعراضنا للتعبير الحسابي لوجدنا أنه معاكس في الاتجاه لعملية بناء هذا التعبير، فلعلك تذكر أن أسلوب بناء التعبير الحسابي وفق النظام التبعي (postifix) وفي ضوء خوارزمية التحويل التي ناقشناها في الجزء السابق، قد سار على نهج الإضافة إلى المكدسات والطوابير. إذ كنا نضيف الرموز إلى نهاية التعبير. أما الآن فإننا نقوم باستعراض هذا التعبير التبعي من اليسار إلى اليمين،

وبذلك فإن عملية الحذف تتم في الاتجاه المعاكس لعملية الإضافة وهذا هو الأسلوب المتبع في الطوابير. ومعنى ذلك أن عمليتي بناء التعبير التبعي وتقييمه تسيران وفق النهج الذي تسير عليه عمليتا الإضافة والحذف في الطوابير. وهذا هو تفسير الملاحظة التي أشرنا إليها في مقدمة هذه الوحدة، عند الحديث عن دواعي الجمع بين الطوابير والمكدسات.

والآن، بعد أن أوضحنا كيفية القيام بعملية التقييم، فإننا قد وصلنا إلى نقطة تستدعي تقديم التفاصيل البرمجية الدقيقة لهذه العملية. ولكن قبل عرض هذه التفاصيل على شكل خوارزمية، دعنا نعطي أولاً التعريفات البرمجية للمتغيرات العامة الضرورية وهي كما يلي:

```
const          n = 20;
typedef char          charbuff[20];
     int m, p ;
     charbuff infix,  postfix;
typedef int StDataType;
 class AStack {
  private:
   char data[n];
    int top;
  public:
              ...
};
```

أما الخطوات التي تنطوي عليها عملية التقييم فهي مضمنة في الخوارزمية "expevaluate" والخوارزمية المصاحبة لها "evaluate" وهما كما يلي:

```
Const                  n=20;
Typedef char                    charbuff[20];
      int m, p ;
      charbuff infix,  postfix;
typedef int StDataType;
  class AStack {
    private:
     char data[n];
     int top;
   public:
     ...
};
```

الخوارزمية (6):

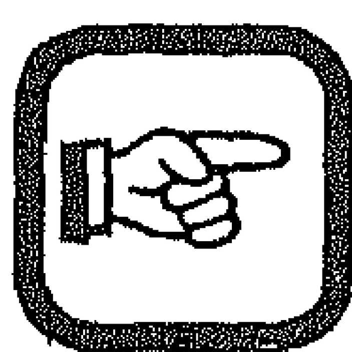
```
void AStack::expevaluate(charbuff postfix,int p,int& result)
 { // evaluating a postfix arithmetic expression
  AStack s;
  int i,val1, val2, charval;
  stackCreate(s);
  for(i=0 ;i<p;i++)
   if(postfix[i]=='^'||postfix[i]=='*'||
     postfix[i]=='/'||postfix[i]=='+'||
     postfix[i]=='-')
     {//the process of evaluation
     pop(s,val1);
     pop(s,val2);
     evaluate(result,val2,postfix[i], val1);
     push(s, result);
     }
    else // transform character into integer
           {charval=i;
      push(s,charval);
      }
    pop(s,result);
 }
```

الخوارزمية (7):

```
void AStack::evaluate(int& result,int val2,char oper,int val1)
 {// evaluating a two-operand arithemtic expression
  switch (oper)
    {case '^'   :result=pow(val2,val1);  break;
     case '*'   :result=val1*val2;  break;
     case '/'   :{if(val1!=0)result=(val2/val1);
                    else result=0;
                 }                                      break;
     case '+'   :result=val1+val2;   break;
     case '-'   :result=val2-val1;   break;
     }
 }
```

مما ينبغي ملاحظته، عزيزي الدارس، في دراستك لهاتين الخوارزميتين هو أن محتويات المصفوفة (postfix) لم تتضمن إلا رموزاً مفردة، وهذا عائد إلى طبيعة تعريفها الذي يحدد

نوع البيانات بالنمط "char". ومعنى ذلك أن هذه المحتويات إما أن تكون رموزاً تمثل العمليات الحسابية التي حددناها من قبل وهي: (^، *، /، +، -) أو رموزاً تعبر عن معاملات. وهذه المعاملات ترد على هيئة أرقام تتراوح بين الصفر والتسعة [0..9]. وبذلك فإن الأمر يقتضي أن نقوم بتحويل هذه الرموز من النمط "char" إلى النمط "int". وهذا ما قمنا به بالفعل في الخوارزمية (6) من خلال المعادلة: int(postfix[i]). ولو أردنا أن نتعامل مع رموز تزيد على عدد واحد، ومع عمليات أكثر، ومع قيم حقيقية وصحيحة معاً لاحتاج الأمر منا إلى معاملة خاصة ودوال أكثر مما عرضناه في هذه الخوارزمية. والهدف المنشود من هذه الخوارزمية هو فقط أن نوضح كيفية استخدام المكدسات لتقييم التعابير الحسابية. وفيما يلي نورد مثالاً على طريقة عمل هذه الخوارزمية.



مثال (3)

افترض أن لدينا التعبير التالي: "5,6,+,*,8,4,/,-" ونرغب في تقييم هذا التعبير والحصول على النتيجة. (لاحظ أن الفارزة ليست جزءاً من التعبير الحسابي. والهدف منها هو الفصل بين الرموز ليس أكثر) ووفقاً للخوارزمية (6)، فإن خطوات التقييم تسير على النحو التالي:

1 - s.top = -1

2 - i = 0

3 - postfix [i] == Operator ? والإجابة بالنفي.

4 - الرمز يمثل أحد المعاملات، وبذلك نقوم بتحويله إلى قيمة عددية ووضعه على المكدس كما يلي:

charval = int(postfix[5])=5

s.data [0] = charval

5 - i = 1

6 - postfix [i] == operator ? والإجابة بالنفي.

7 -الرمز يمثل أحد المعاملات، وبذلك نقوم بتحويله إلى قيمة عددية ووضعه على المكدس:

charval = int(postfix[6])=5

s.data [1] = charval

8 - i = 2

9 - postfix [i] == operator ? والإجابة بالنفي.

10 -الرمز يمثل أحد المعاملات، وبذلك نقوم بتحويله إلى قيمة عددية ووضعه على المكدس:

charval = int(postfix[2])=2

s.data [2] = charval

11 -i = 3

12 -? postfix [i] == operator والإجابة بالإثبات.

اذن: val1 = 2, val 2 = 6

result = val2 + val1

s.data [1] = result = 8

13 -i =4

14 - ? postfix [i] == operator والإجابة بالإثبات.

إذن: val1 = 8, val2 = 5

result = val2 * val1

s.data [1] = result = 40

15 -i =5

16 - ? postfix [i] == operator والإجابة بالنفي.

17 -الرمز يمثل أحد المعاملات، وبذلك نقوم بتحويله إلى قيمة عددية ووضعه على المكدس:

charval = int(postfix[8])=8

s.data [2] = charval

18 -i =6

19 - ? postfix [i] == operator والإجابة هي بالنفي.

20 -الرمز يمثل أحد المعاملات، وبذلك تقوم بتحويله إلى قيمة عددية وضعه على المكدس:

charval = int(postfix[4])=4

s.data [3] = charval

21 -i =7

22 -? postfix [i] == operator والإجابة بالإثبات.

إذن: val1 =4, val2 =8

result = val2/ val1

s.data [1] = result = 2

23 -i =8

24 -? postfix [i] == operator والإجابة بالإثبات.

إذن: val1 =2, val2 =40

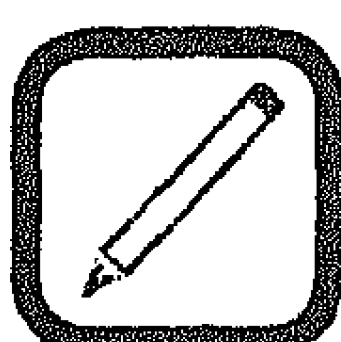
result = val2- val1

s.data [1] = result = 38

25 -لقد أصبحت قيمة "i" مساوية لقيمة "p"، وبذلك تنتهي عملية التقييم (انظر الجدول (6)، والنتيجة إذن هي: "result = 38".

جدول (6): وضع المكدس خلال المراحل المختلفة لتقييم تعبير حسابي

	POSTFIX	STACK
1	5	5
2	6	5,6
3	2	5,6,2
4	+	5,8
5	*	40
6	8	40,8
7	4	40,8,4
8	/	40,2
9	-	38



تدريب (6)

تأمل الخوارزمية (6) والمثال (3) الذي بني عليها مرة أخرى، ثم قم بعملية تقييم للتعبير الحسابي التالي المعطى بصيغة النظام التبعي (postfix) (لاحظ أن الهدف من الفارزة هو فقط الفصل بين الرموز وتجنب اللبس): "^ ,- ,5 ,7 ,* ,2 ,+ ,8 ,2"

يفضل أن تتكون إجابتك على هذا التدريب من جزئين: الأول هو تسلسل الخطوات على النحو الوارد في المثال (3)، والثاني هو توضيح ذلك بالرسم على النحو الوارد في الجدول (6).

تدريب (7)



أجب بنعم أو لا وعلل إجابتك:

أ- إن نتيجة تقييم التعبير التالي: "+,+,*,*,-,2,3,4,5,6,7" هي "74".

ب- إن نتيجة تحويل التعبير التالي: "(6-4) * (3+2) " إلى النظام القبلي (prefix) هي: "*,+,2,3,-,6,4".

ج- ان نتيجة تحويل التعبير التالي: " (A+B) * C" إلى النظام التبعي (postfix) هي: "A B + C*".

د- ان نتيجة تحويل التعبير التالي: "AB+CD-*" إلى النظام الوسطي (infix) هي: "(A+B)*C-D"

هـ- ان نتيجة تحويل التعبير التالي:"(A+B*C)-D" إلى النظام التبعي هي: "ABCD+*-".

5. الخلاصة

المكدسات هي نوع خاص من القوائم حيث تتم الإضافة (push) والحذف (pop) من نفس الجانب، مما يجعلها تركيبة بيانات مناسبة للكثير من التطبيقات التي تتطلب التعامل مع البيانات حسب مبدأ من يصل آخراً يغادر أولاً (ليفو LIFO).

وهنالك العديد من التطبيقات التي تتطلب مثل هذا الترتيب في الخدمة منها عملية استدعاء البرامج الفرعية، وتحويل التعابير الرياضية من الحالة الوسطية Infix إلى الحالة التبعية postfix. هذا بالإضافة إلى عملية حساب التعابير الرياضية التبعية.

ومن العمليات التي تجرى على المكدسات بالإضافة إلى عمليتي الإضافة والحذف، عمليات إنشاء مكدس StackCreate، والتأكد فيما إذا كان المكدس ممتلئاً StackFull، والتأكد فيما إذا كان المكدس فارغاً (خالياً) StackEmpty. وهنالك طريقتان رئيستان لتمثيل المكدسات: التمثيل التتابعي باستخدام المصفوفات والتمثيل المتصل باستخدام القوائم المتصلة والمؤشرات.

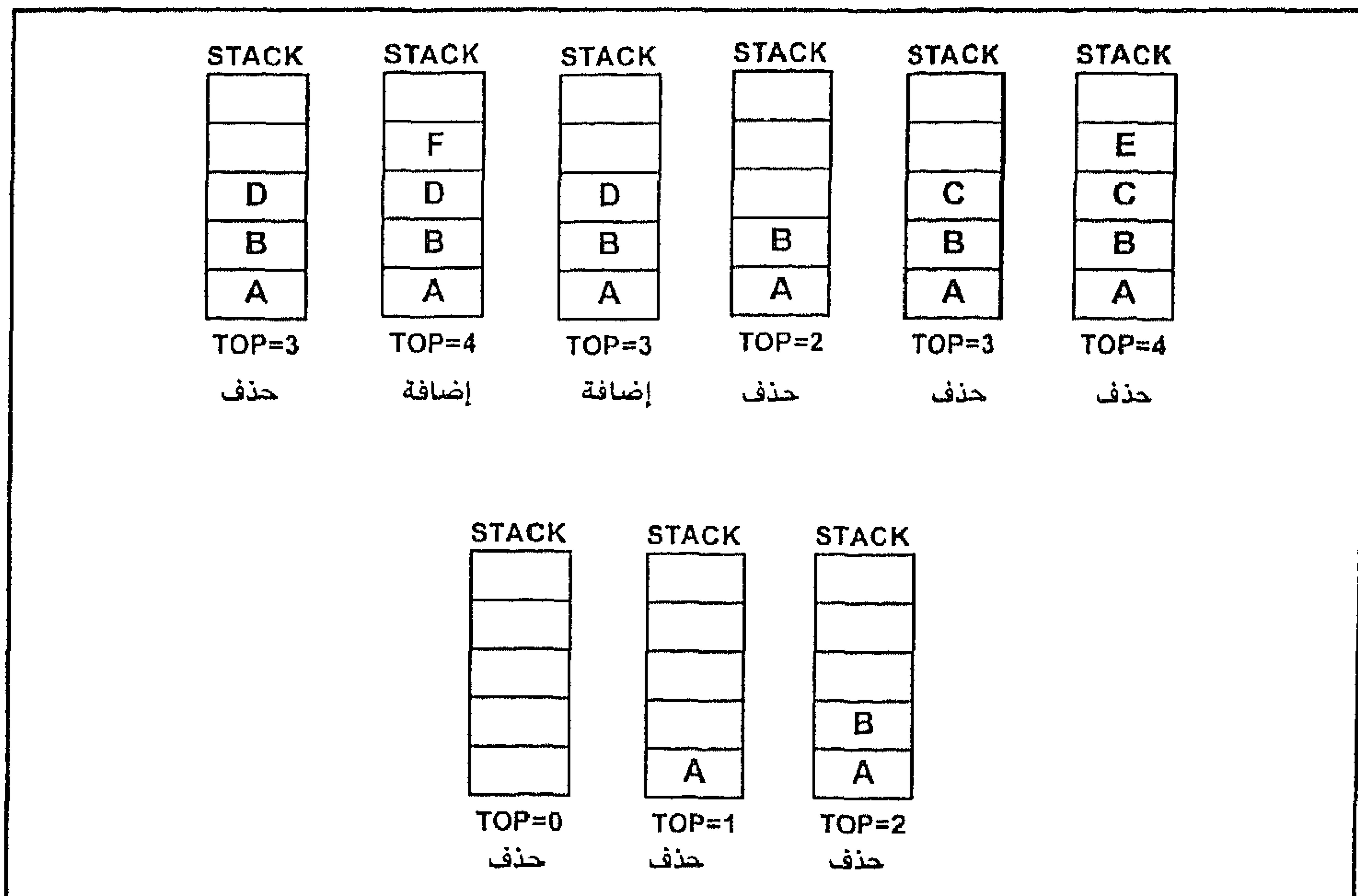
6. لمحة عن الوحدة الدراسية الثامنة

في الوحدة التالية، عزيزي الدارس، سنقوم بمناقشة واحد من أهم تطبيقات المكدسات وهو استخدامها لتنفيذ البرامج الفرعية التي تستدعي نفسها ذاتياً (Recursive Programs).

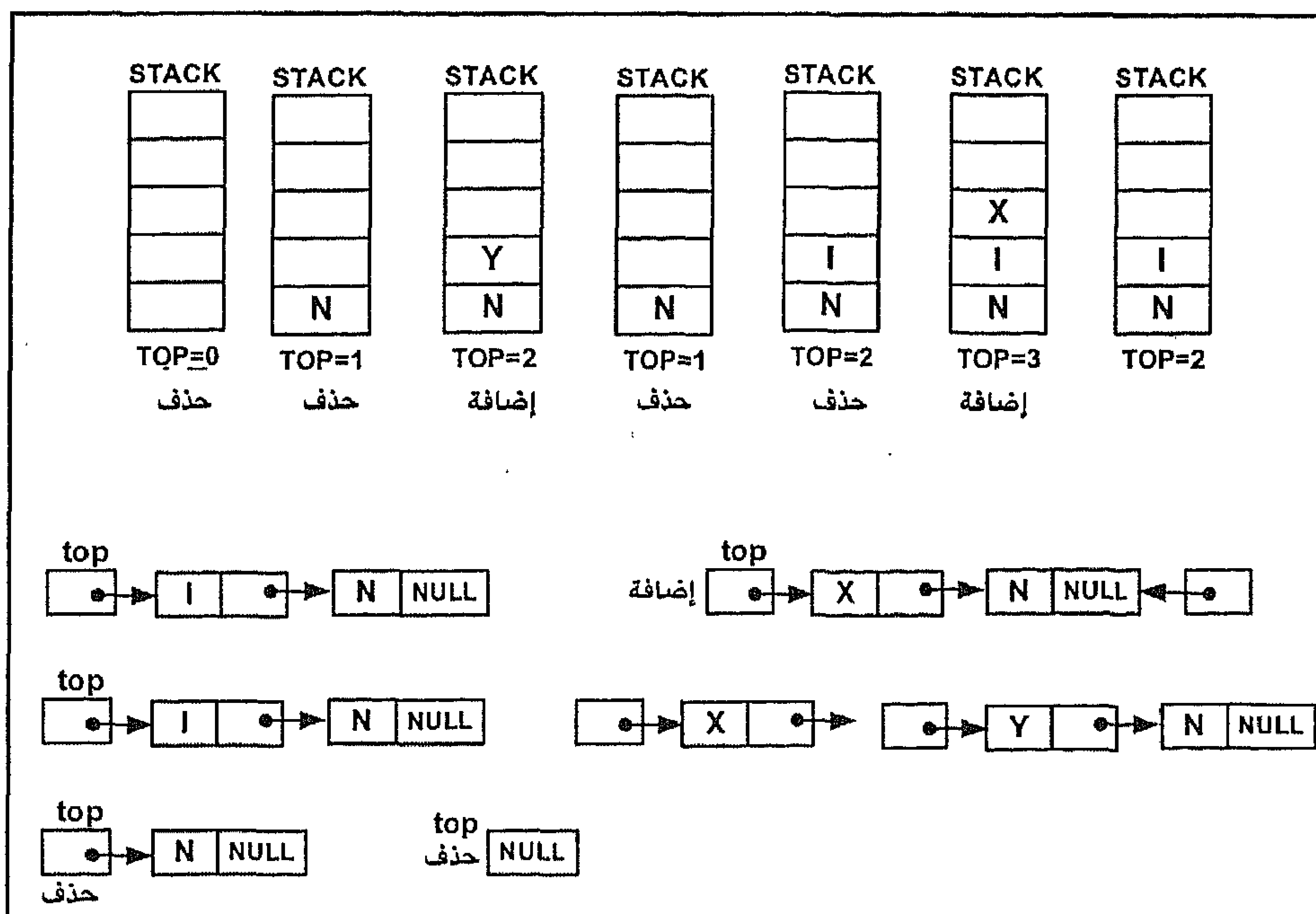
7. إجابات التدريبات

تدريب (1)

STACK	STACK	STACK	STACK	STACK	STACK
G					
E	E				
C	C	C			
B	B	B	B		
A	A	A	A	A	
TOP=5	TOP=4	TOP=3	TOP=2	TOP=1	TOP=0
إضافة	إضافة	إضافة	إضافة	إضافة	خالي


STACK
D
B
A
TOP=3
حذف
STACK
F
D
B
A
TOP=4
إضافة
STACK
D
B
A
TOP=3
إضافة
STACK
B
A
TOP=2
حذف
STACK
C
B
A
TOP=3
حذف
STACK
E
C
B
A
TOP=4
حذف
STACK
TOP=0
حذف
STACK
A
TOP=1
حذف
STACK
B
A
TOP=2
حذف

تدريب (2)


STACK
TOP=0
حذف
STACK
N
TOP=1
حذف
STACK
Y
N
TOP=2
إضافة
STACK
N
TOP=1
حذف
STACK
I
N
TOP=2
حذف
STACK
X
I
N
TOP=3
إضافة
STACK
I
N
TOP=2
top
I
N
NULL
top
إضافة
X
N
NULL
top
I
N
NULL
X
Y
N
NULL
top
N
NULL
حذف
top
حذف
NULL

تدريب (3)

```
const n=26;
typedef char StckData;
typedef char str[n];
str alpha;
......
void PtrStack::reverse(str& alpha)
  { //var     top: stackptr;
   //SPtr top;
   int i;
   StckData element;
   for(i=1;i<=n;i++)
    push(alpha[i]);
   for(i=1;i<=n;i++)
    {pop(element);
     cout<<" element";
  }
```

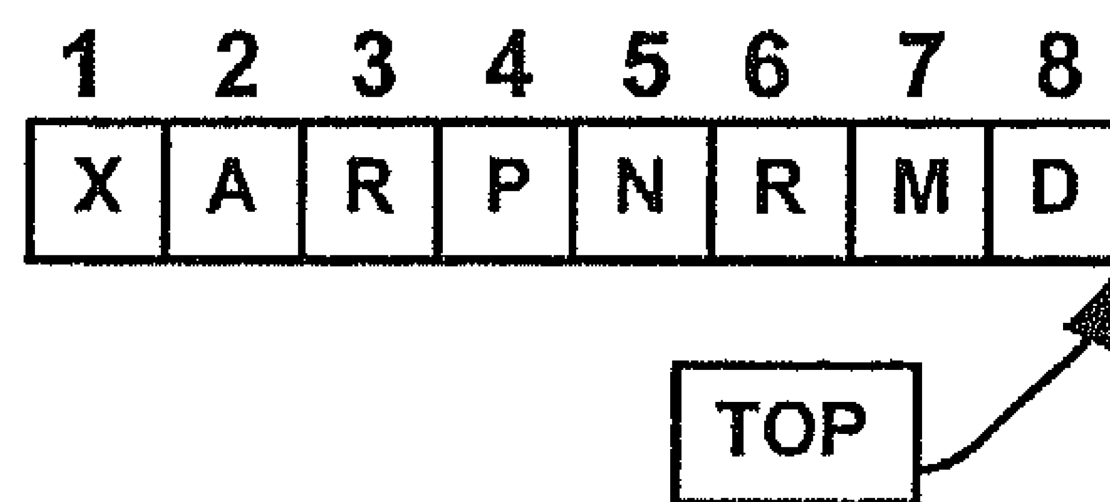
تدريب (4)

أ- top = 4

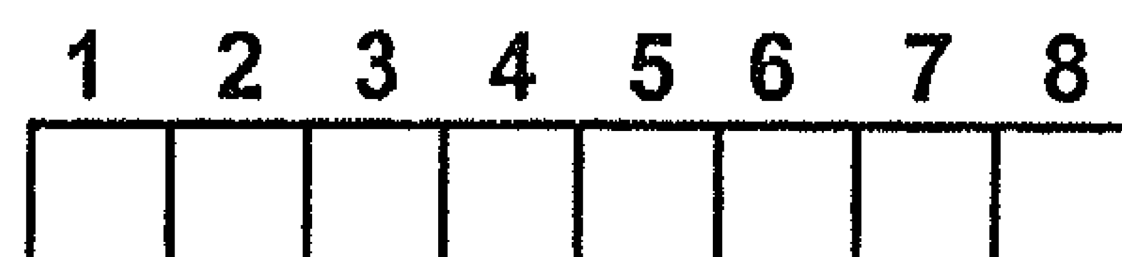
ب- n = 8

ج- top = 0

د-



هـ-



و- top = 0

تدريب (5)

	infix	stack	postfix
1	A		A
2	+	+	A
3	(	+(	A
4	B	+(	AB
5	*	+(*	AB
6	C	+(*	ABC
7	-	+(-	ABC *
8	(	+(-(	ABC *
9	D	+(-(	ABC *D
10	/	+(-(/	ABC *D
11	E	+(-(/	ABC *DE
12	^	+(-(/^	ABC *DE
13	F	+(-(/^	ABC *DEF
14	)	+(-	ABC *DEF^/
15	*	+(- *	ABC *DEF^/
16	G	+(- *	ABC *DEF^/G
17	)	+	ABC *DEF^/G *-
18	*	+ *	ABC *DEF^/G *-
19	H	+ *	ABC *DEF^/G *-
			ABC *DEF^/G *-+

تدريب (6)

	postfix	stack
1	2	2
2	8	2,8
3	+	10
4	2	10,2

5	*	20
6	7	20,7
7	5	20,7,5
8	-	20,2
9	^	400

تدريب (7)

أ- نعم! وخطوات الحل هي كما يلي:

exp	stack
2	2
3	2, 3
4	2, 3, 4
5	2, 3, 4, 5
6	2, 3, 4, 5, 6
7	2, 3, 4, 5, 6, 7
-	2, 3, 4, 5, 1
+	2, 3, 4, 6
*	2, 3, 24
*	2, 72
+	74

ب- نعم! وخطوات التحويل هي كما يلي:

1. +, 2, 3
2. -, 6, 4
3. *, +, 2, 4, -, 6, 4

ج- نعم! وخطوات التحويل هي كما يلي:

1. A, B, +
2. C
3. A, B, +, C, *

د- لا! وخطوات التحويل هي كما يلي:

1. (A + B)

2. (C-D)

3. (A+B)*(C-D)

هـ- لا! وخطوات التحويل هي كما يلي:

1. A
2. B, C, *.
3. A, B, C, *, +.
4. D.
5. A, B, C, *, +, D, -.

8. مسرد المصطلحات

- **الإضافة (الدفع) Push**: تخزين قيمة جديدة في المكدس وذلك من خلال قمة المكدس.

- **تعابير رياضية (حسابية) تبعية Postfix Expression**: تعبير يعتمد على وضع رمز العملية الحسابية بعد المعاملين المقصودين بالعملية (مثال ذلك: AB+)

- **تعابير رياضية (حسابية) قبلية Prefix Expression**: تعبير يعتمد على وضع رمز العملية الرياضية قبل المعاملين اللذين ينتميان إليه (مثال ذلك: AB+).

- **تعابير رياضية (حسابية) وسطية Infix Expression**: تعبير يعتمد على وضع الرمز المعبر عن أحد العمليات الحسابية بين المعاملين اللذين ينتميان إليه (مثال ذلك: A+B)

- **الحذف (الإطلاق) Pop**: حذف قيمة موجودة في المكدس وذلك من خلال قمة المكدس.

- **المكدس Stack**: تركيب بياني عبارة عن قائمة من القيم مخزنة بأسلوب معين من أساليب تمثيل القوائم الذي يحاكي السلوك الواقعي ويعمل وفق مبدأ: "من يأتي آخراً يغادر أولاً"، أي أنّ آخر قيمة تضاف إلى المكدس هي أول قيمة تحذف منه.

9. المراجع

1. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice -Hall. 2000.
2. Tremplay, J.P.; and Sorenson, P.G.; An Introduction to Data Structures with Applications, 2nd Edition, McGraw-Hill, 1984.
3. Amsbury, Wagne, Data Structures from Arrays to Priority Queues. Belmont (USA): Wadsworth, 1985.
4. Kruse, Robert L., Data Structures and Program Design. Englewood Cliffs (USA): Prentice-Hall, 1984.
5. Lipschutz, Seymour, Theory and Problems of Data Structures. New York: McGraw-Hill, 1986.
6. Miller, Lawrence H., Advanced Programming Design and Structures. New York: McGraw-Hill, 1986.
7. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, 2nd Edition, Addison Wesley, 1999.
8. Lewis, T. G.; Smith, M.Z., Applying Data Structures. Atlanta (USA): Houghton Mifflin, 1976.
9. Tenenbaum, Aarom M.; Augenstein, Moshe J., Data Structures using Pascal. Englewood Cliffs (USA): Prentice-HALL, 1981.

الوحدة
الثامنة

الاستدعاء الذاتي Recursion

محتويات الوحدة

1. المقدمة

1.1 التمهيد

أهلاً بك، عزيزي الدارس، إلى الوحدة الثامنة من كتاب "تركيب البيانات وتصميم الخوارزميات"، وهي بعنوان "الاستدعاء الذاتي". لا شك أنك تعلم، عزيزي الدارس، أن العديد من لغات البرمجة كلغة C/C++ ولغة جافا تسمح للبرامج الفرعية (Subprograms) أن تستدعي نفسها استدعاءً ذاتياً، مما يعطي هذه اللغات خاصية تجعلها مناسبة لبرمجة العديد من المسائل بسهولة ويسر. سينصب اهتمامنا، في هذه الوحدة على الجوانب المختلفة المتعلقة بالاستدعاء الذاتي الذي يعتبر واحداً من أكثر تطبيقات المكدسات أهمية كما سنرى.

إذ سنناقش أهمية الاستدعاء الذاتي وكيف ومتى نستخدمه ثم سنناقش الآلية التي ينفذ بها الحاسوب الاستدعاء الذاتي ودور المكدسات في هذه الآلية. ثم ستناقش كيفية محاكاة البرامج الجزئية التي تستخدم الاستدعاء الذاتي.

نود أن نلفت انتباهك، عزيزي الدارس، إلى أهمية هذا الموضوع إذ أن هنالك العديد من المسائل التي تبدو معقدة يسهل حلها إذا أتقن المرء فن كتابة البرامج التي تستدعي نفسها استدعاءً ذاتياً. وسنرى العديد من هذه المسائل في هذه الوحدة والوحدات القادمة في هذا الكتاب.

2.1 أهداف الوحدة

ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1. تميّز المسائل التي يتطلب حلها استخدام الاستدعاء الذاتي.
2. تكتب البرامج الفرعية التي تستخدم الاستدعاء الذاتي.
3. توضح الآلية التي يتم فيها تنفيذ برامج الاستدعاء الذاتي ودور المكدسات في هذه الآلية.
4. تحوّل البرنامج الفرعي الذي يستخدم الاستدعاء الذاتي إلى برنامج فرعي لا يستدعي نفسه ولكن يقوم باستخدام المكدسات عوضاً عن ذلك.

3.1 أقسام الوحدة

إن الأقسام الثلاثة الرئيسة المكونة لهذه الوحدة تتعلق بتحقيق الأهداف المشار إليها أعلاه. ففي القسم الأول سنبين كيفية تمييز المسائل التي يتطلب حلها استخدام الاستدعاء الذاتي وكيفية كتابة تلك البرامج، ويرتبط هذا القسم بالهدفين الأول والثاني. وفي القسم

الثاني سنناقش الآلية التي تنفذ بها البرامج الفرعية التي تستدعي نفسها ودور المكدسات بها، ويرتبط هذا القسم بالهدفين الثالث والرابع. وفي القسم الثالث نناقش مسألة من المسائل التي يصعب حلها دون استخدام الاستدعاء الذاتي وهي "أبراج هانوي" ويرتبط هذا القسم بجميع الأهداف الأربعة.



4.1 القراءات المساعدة

على الرغم من شمولية الأقسام والمسائل التي تم عرضها في هذه الوحدة إلا أن هناك فوائد كثيرة يمكن أن يجنيها الدارس من الرجوع إلى بعض القراءات الإضافية المساعدة. فهناك المزيد من المعلومات والأمثلة في هذه المصادر، وهناك أيضاً المزيد من التدريبات والأسئلة. ونوصي بالمصدرين التاليين لهذه الغاية، مع التذكير بأن قائمة المراجع في نهاية هذه الوحدة تتضمن مصادر على قدر كبير من الأهمية والفائدة:

1. Amsbury, Wayne, Data Structures from Arrays to Priority Queues. Belmont (USA): Wadsworth, 1985. pp. 97-169 & ,141-120 ,119-198.

2. Malik, D.S. Data Structures Using C++, 1st Edition, Course Technology, Inc., 2003.

3. Main, Michael Data Structures & Other Objects Using C++, 3d Edition, Addison-Wesley, 2004.

2. كيف ومتى نستخدم الاستدعاء الذاتي؟

هنالك العديد من المسائل التي يتطلب حلها استخدام الاستدعاء الذاتي وغالباً ما يكون تعريف هذه المسائل نفسها تعريفاً ذاتياً (Recursive)، على سبيل المثال يعرف مضروب العدد الصحيح !n تعريفاً ذاتياً كما يلي:

$$n! = \begin{cases} 1 & f \quad n = 0 \\ n * (n-1)! & if n > 0 \end{cases}$$

أي أن مضروب الصفر هو الرقم 1 ومضروب أي رقم صحيح أكبر من الصفر يساوي حاصل ضرب ذلك الرقم في مضروب الرقم الذي يقل عنه بواحد.

لاحظ أن ذلك التعريف هو تعريف ذاتي بمعنى أننا عرفنا دالة المضروب مستخدمين دالة المضروب نفسها. ولهذا يعتبر المضروب من المسائل الشهيرة التي يسهل حلها باستخدام الاستدعاء الذاتي.

والبرنامج الفرعي التالي يبين كيفية إيجاد المضروب باستخدام الاستدعاء الذاتي:

```
int fact(int n)
  {if (n==0)
    n=1;
    else
     n=n*fact(n-1);
   return n;
  }
```

لاحظ، عزيزي الدارس، مدى توافق الدالة fact مع تعريف المضروب، مما يجعلها سهلة الكتابة والفهم.

لاحظ أن الجملة:

```
n=n*fact(n-1);
```

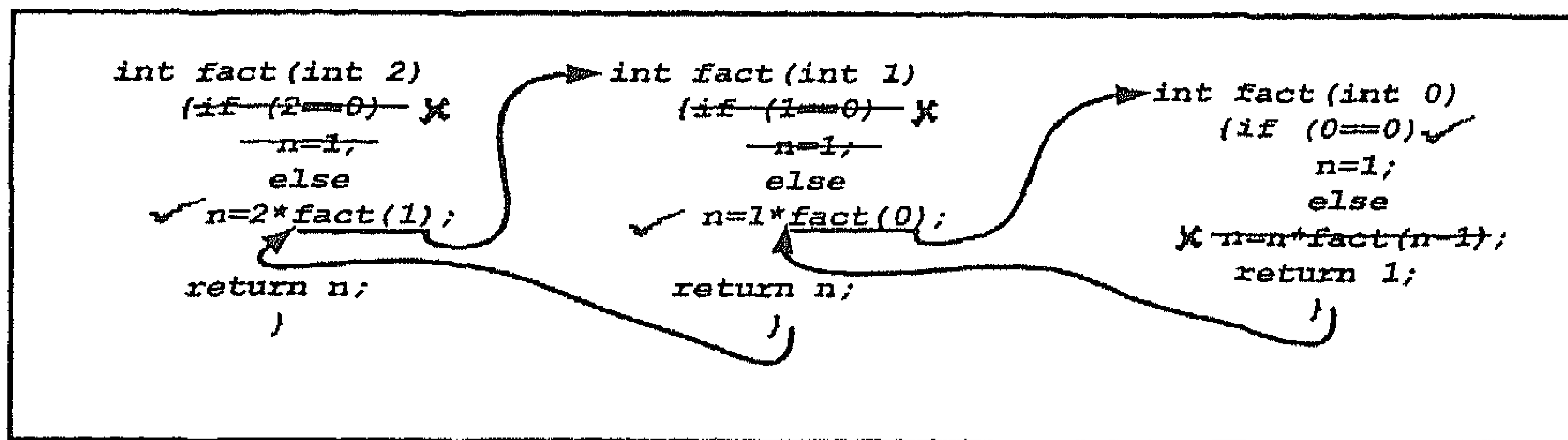
هي الجملة التي تستدعي الدالة استدعاءً ذاتياً لحساب مضروب n-1.

وسندرس في قسم لاحق كيف تنفذ هذه الدالة بالتفصيل. ويكفي لفهم ما يجري في هذه المرحلة أن تتخيل، أن الحاسوب يعمل نسخة من هذه الدالة عند كل مرة يستدعي فيها fact وينفذ تلك النسخة ثم يعود الحاسوب لإكمال تنفيذ النسخة المستدعية كأي برنامج فرعي آخر إذ تعود السيطرة (Control) عند انتهاء تنفيذ البرنامج الفرعي إلى البرنامج المستدعي له. والشكل (1) يبين كيفية إيجاد المضروب للرقم 2 مستخدمين fact على افتراض أن الحاسوب يقوم بعمل نسخة جديدة في كل مرة يستدعي فيها البرنامج نفسه ذاتياً.

لاحظ، عزيزي الدارس، أن مسألة المضروب تتميز بالميزتين التاليتين:

أولاً: هنالك حالة لا يتطلب حساب المضروب لها استدعاءً ذاتياً وهي عند n=0 تسمى هذه الحالة القاعدة أو الأساس.

ثانياً: في كل مرة تُستدعى فيها الدالة استدعاءً ذاتياً تُستدعى على حالة أبسط من الحالة الأصلية بمعنى أنها أقرب إلى الحالة الأساس من الحالة الأصلية، إذ أننا في كل مرة نستدعي فيها fact نستدعيه على رقم أقل بواحد من الرقم الأصلي وذلك ضروري لضمان أننا سنصل في نهاية المطاف إلى الحالة الأساس التي تتوقف عندها عملية الاستدعاء الذاتي ولولا ذلك فقد لا تتوقف عملية الاستدعاء الذاتي وتصبح عملية الاستدعاء لانهائية.



الشكل (1): كيفية حساب fact(2) على افتراض أن الحاسوب يقوم بنسخ البرنامج الفرعي عند كل مرة يُستدعى فيها. لاحظ أن (×) تعني أن الجملة المقابلة لا تنفذ لعدم تحقق الشرط وأن (✓) تعني أن الجملة المقابلة ستنفذ لتحقق الشرط

إن من الضروري توفر هاتين الميزتين في أي مسألة كي يكون حلها بالاستدعاء الذاتي ممكناً أي يجب أن:

1. تتوفر حالة (أو أكثر) أساس لا يتطلب حلها استدعاءً ذاتياً.
2. في كل مرة يستدعى فيها البرنامج الفرعي استدعاءً ذاتياً يستدعى على حالة أبسط (أي أقرب إلى الحالة الأساس) من الحالة الأصلية.

إن توفر هذين الشرطين ضروري جداً لضمان توقف البرنامج أي كي لا يصبح الاستدعاء الذاتي لانهائي. وكمثال آخر على الاستدعاء الذاتي لندرس المتتالية التالية:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

تدعى هذه المتتالية متتالية فابونانشي حيث أن كل رقم بها باستثناء أول رقمين يساوي حاصل جمع الرقمين السابقين أي أن الحد رقم n ((fib(n)) يحسب كما يلي:

fib(n)=fib(n-1)+fib(n-2)   for n>2

وسنكتب برنامجاً فرعياً (fib)، يحدد لنا قيمة الحد رقم n، أي إذا استدعينا fib(7) فإنه يحسب لنا قيمة الحد السابع في المتتالية وهي «8». والسؤال هنا هل من الممكن حل هذه المسألة باستخدام الاستدعاء الذاتي؟ وللإجابة على هذا السؤال يجب أن يتحقق الشرطين السالف ذكرهما:

1. هل هنالك حالة (أو أكثر) لا يتطلب حلها استدعاءً ذاتياً، والجواب هو بالإيجاب إذ أن حساب العنصر الأول والثاني لا يتطلب استدعاءً ذاتياً.
2. هل يستدعى البرنامج الفرعي استدعاءً ذاتياً بحالة أبسط من الحالة الأصلية؟ والجواب أيضاً على هذا السؤال إيجابياً إذ لحساب fib(n) يستدعى fib لحساب fib(n-1) ولحساب fib(n-2). أما الدالة نفسها فهي كما يلي:

```
int fib(int n)
{int x,y;
if (n==1)
n=0;
else
if (n==2)
n=1;
else
{x=fib(n-1);
y=fib(n-2);
n=x+y;
}
return n;
}
```

3. كيفية تنفيذ برامج الاستدعاء الذاتي

سبق وذكرنا، عزيزي الدارس، أنه من الممكن تصور أن ما يقوم به الحاسوب عند تنفيذ برنامج هو عمل نسخة جديدة من ذلك البرنامج الفرعي عند كل مرة يستدعي فيها نفسه ثم تنفذ تلك النسخة، وعند الانتهاء تعاد السيطرة إلى النسخة المستدعية وهكذا. ولو كان هذا فعلاً ما يجري لتطلب الأمر مساحة كبيرة من ذاكرة الحاسوب قد لا تتوفر دائماً. لحسن الحظ لسنا بحاجة إلى عمل نسخة جديدة من البرنامج الفرعي في كل مرة يُستدعى فيها استدعاءً ذاتياً، إذ يكفي الاحتفاظ بنسخ عن قيم بعض المتغيرات والعوامل، وعلى عنوان الجملة التي يجب العودة إليها واستكمال تنفيذ البرنامج الفرعي وذلك عند الانتهاء من تنفيذ جملة الاستدعاء الذاتي للبرنامج الفرعي.

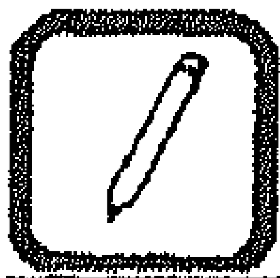
لنعد إلى دراسة البرنامج الفرعي (fact) الذي يحسب مضروب عدد صحيح ما، لاحظ أننا من الممكن أن نكتفي بتخزين قيمة n الحالية وعنوان الجملة التي تقوم بعملية الضرب وهي الجملة التي يجب أن يستكمل تنفيذ للبرنامج الفرعي عندها، وذلك بعد الانتهاء من تنفيذ عملية الاستدعاء الذاتي. على سبيل المثال لحساب fact(2) يجب حساب fact(1) وبعد الانتهاء من حساب fact(1) تضرب قيمته بقيمة "n" وهي "2"، وعليه يجب حفظ قيمة "n" (وهي 2) قبل الانتقال إلى تنفيذ fact(1) كي تستخدم في عملية الضرب عند الانتهاء من حساب fact(1) . ولحساب fact(1) يجب حساب fact(0) وعند الانتهاء من ذلك يجب العودة وضرب قيمة fact(1) بقيمة "n" وهي "1" للحصول على قيمة fact(1) وعليه يجب تخزين قيمة "n " وهي "1" قبل الانتقال لحساب fact(0).

لاحظ، عزيزي الدارس، أن ترتيب تخزين قيم "n" هو خزن قيمة n=2 ثم خزن قيمة n=1 ولكن ترتيب استخدامها في عملية الضرب هو العكس تماماً إذ أن آخر قيمة خزنت لـ n هي أول قيمة تستخدم في عملية الضرب وأول قيمة خزنت لـ n هي آخر قيمة تستخدم في عملية الضرب. وعليه فإنه من المناسب استخدام مكدس (stack) لتخزين قيم n المختلفة.

ولاستيعاب كيفية استخدام المكدس سوف ندرس الكيفية التي يتغير بها المكدس عند حساب fact(2). وعند البدء يكون المكدس خالياً كما يبدو في الشكل (2-أ) ثم وقبل الانتقال لحساب قيمة fact(1) تخزن قيمة (n=2) في المكدس كما يبدو في شكل (2-ب). وقبل الانتقال لحساب قيمة fact(0) يتم تخزين قيمة (n=1) على المكدس كما يبدو في الشكل (2-ج)، وعند الانتهاء من حساب fact(0) يتم استرجاع قيمة "n " السابقة (للقيام بعملية الضرب) وذلك بعمل عملية شطب من المكدس(pop) ويصبح المكدس كما يبدو في الشكل (2-د)، ثم تنفذ عملية الضرب.

وعليه تنتهي عملية حساب fact(1) ويجب العودة الآن وإكمال حساب fact(2) فيتم استرجاع قيمة(n=2)من المكدس عن طريق عملية شطب (pop) منه فيعود المكدس خالياً كما بدأ (شكل (2-هـ))، ثم تنفذ عملية الضرب لحساب 2*fact(1) وبذلك نحصل على قيمة fact(2).

الشكل (2): الكيفية التي يتغير بها المكدس في أثناء القيام بحساب fact(2)



تدريب (1)

بيّن، عزيزي الدارس، كيف يتغير المكدس في أثناء حساب fact(5).

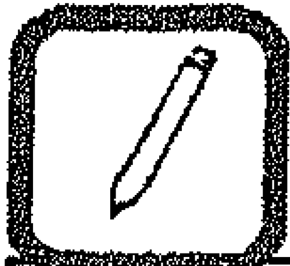
والبرنامج الفرعي التالي fact2 يوضح بشكل تقريبي الخطوات التي يقوم بها الحاسوب عند تنفيذ البرنامج الفرعي (fact):

```
int fact2(int n)
{int f;
 AStack S;
stackCreate(S);
L1:if (n==0)f=1
else
if(!stackFull(S))
{push(S,n);
n--;
goto L1;
}
while(!stackEmpty(S))
{pop(S,n);
f*=n;
}
n=f;
return n;
}
```

إن fact2 يستخدم المكدس (S) للاحتفاظ بقيم "n" المختلفة فهو يقوم بفحص قيمة "n"، فإذا لم تكن صفراً فإنه يخزن قيمة "n" الحالية على المكدس (S) ثم ينتقل لحساب مضروب (n-1) وذلك بطرح"1" من "n" ثم الانتقال إلى أول جملة في البرنامج الفرعي وتكرار نفس الخطوات. وعندما تصبح قيمة "n" صفر فإن المضروب (f) يصبح "1"، والآن يجب ضرب "f" بقيم n"" المكدسة على (S) حتى يفرغ المكدس (S).

ويتم ذلك بشطب (pop) قيمة "n" من المكدس وضرب قيمة "n" بـ (f) حتى يفرغ المكدس S.

سبق وأشرنا، عزيزي الدارس، أن (fact2) يوضح خطوات تنفيذ (fact) بشكل تقريبي جداً إذ أنه لا يخزن على المكدس عنوان الجملة التي يجب أن تعاد السيطرة (control) إليها عند الانتهاء من تنفيذ جملة الاستدعاء ولا يستخدم ذلك العنوان، وذلك لأن الجملة التي يجب العودة اليها هي دائماً جملة الضرب في هذا البرنامج البسيط.



تدريب (2)

هل يجب تخزين عنوان جملة العودة عند محاولة محاكاة البرنامج الفرعي (fib).

ولتوضيح الفكرة بشكل أفضل لندرس البرنامج الفرعي (الدالة) التالي:

```
void reverse()
 {char ch;
  if (endl)
   {cin>>ch;
    reverse();
    cout<<ch;
    }
  }
```

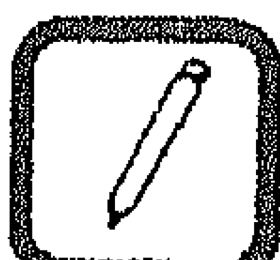
إن ما تقوم به هذه الدالة التي تستدعي نفسها ذاتياً هو قراءة سطر من الرموز(line of characters) وطباعتها بشكل عكسي على سبيل المثال إذا أدخلت السطر التالي:

This is a line of text

تحصل على النتيجة التالية:

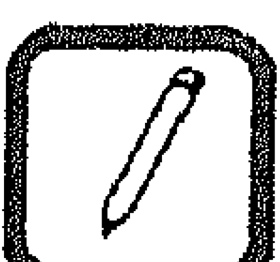
txet fo enil a si sihT

تقوم هذه الدالة بعد التأكد من عدم انتهاء السطر بقراءة رمز ثم تستدعي نفسها استدعاءً ذاتياً وهذا يتطلب الاحتفاظ بقيمة ((ch في مكدس لاستخدامها لاحقاً وعند الانتهاء من تنفيذ جملة الاستدعاء الذاتي يقوم الحاسوب باسترجاع قيمة (ch) من المكدس ثم ينفذ جملة الطباعة (cout) وعليه فإن أول رمز يقرأ هو آخر رمز سوف يكتب.



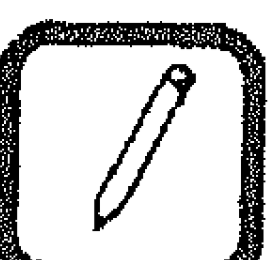
تدريب (3)

ارسم حالات المكدس المختلفة عند تنفيذ (reverse) وإعطائه المدخلات التالية:((text



تدريب (4)

اكتب برنامجاً فرعياً (reverse2) يحاكي بشكل تقريبي ما يقوم به الحاسوب عند تنفيذ (reverse) شريطة أن لا يقوم (reverse2) باستدعاء نفسه.



تدريب (5)

إذا كان بإمكانك، عزيزي الدارس، حل مسألة ما باستخدام برنامج فرعي يستدعي نفسه استدعاءً ذاتياً، وأمكنك أيضا بنفس القدر من السهولة واليسر حل المسألة باستخدام برنامج لا يستخدم الاستدعاء الذاتي، فأيهما تفضل؟

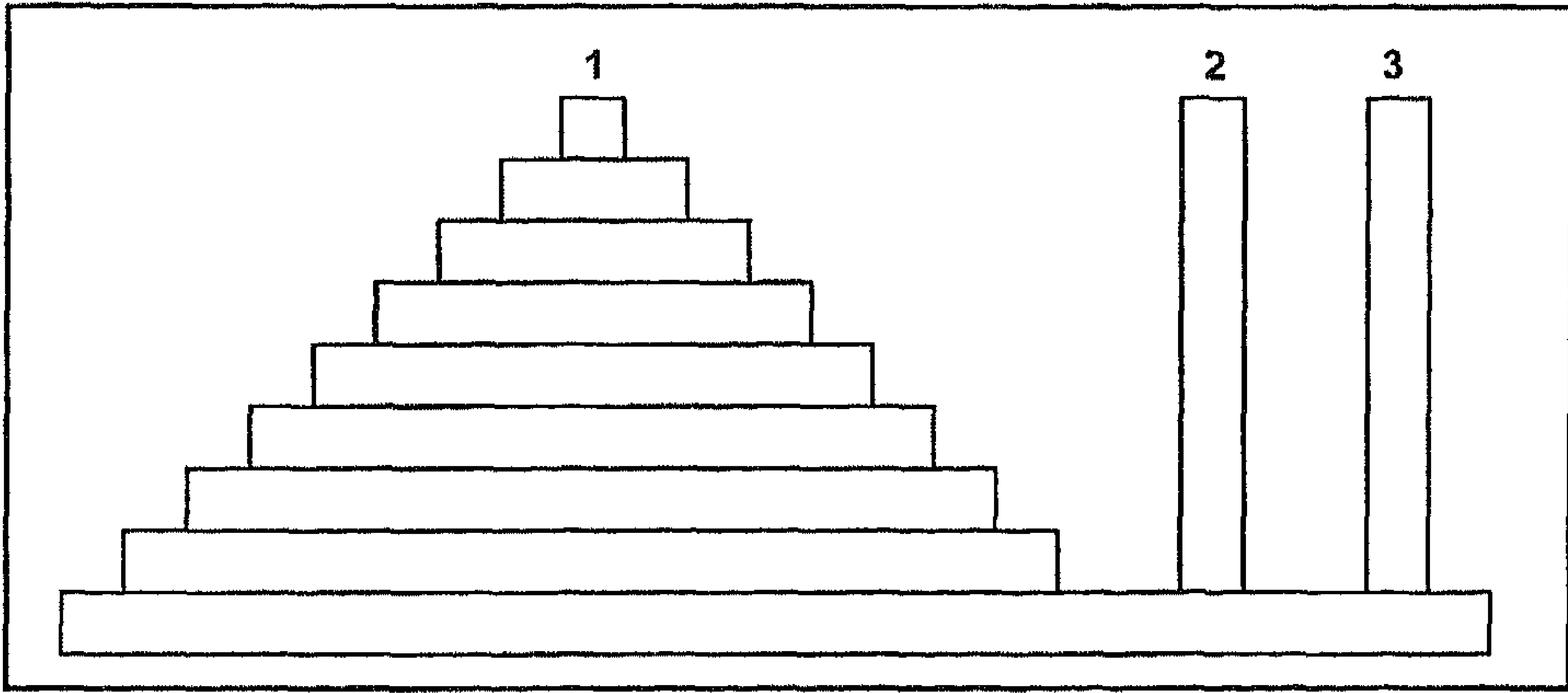


أسئلة التقويم الذاتي (1)

1. ما هي المزايا التي من الضروري توفرها في أي مسألة كي يكون حلها بالاستدعاء الذاتي ممكناً.
2. وضح لماذا من المناسب استخدام مكدس (stack) لتنظيم عملية الاستدعاء الذاتي.
3. ما الفائدة من تخزين عنوان جملة العودة على المكدس.

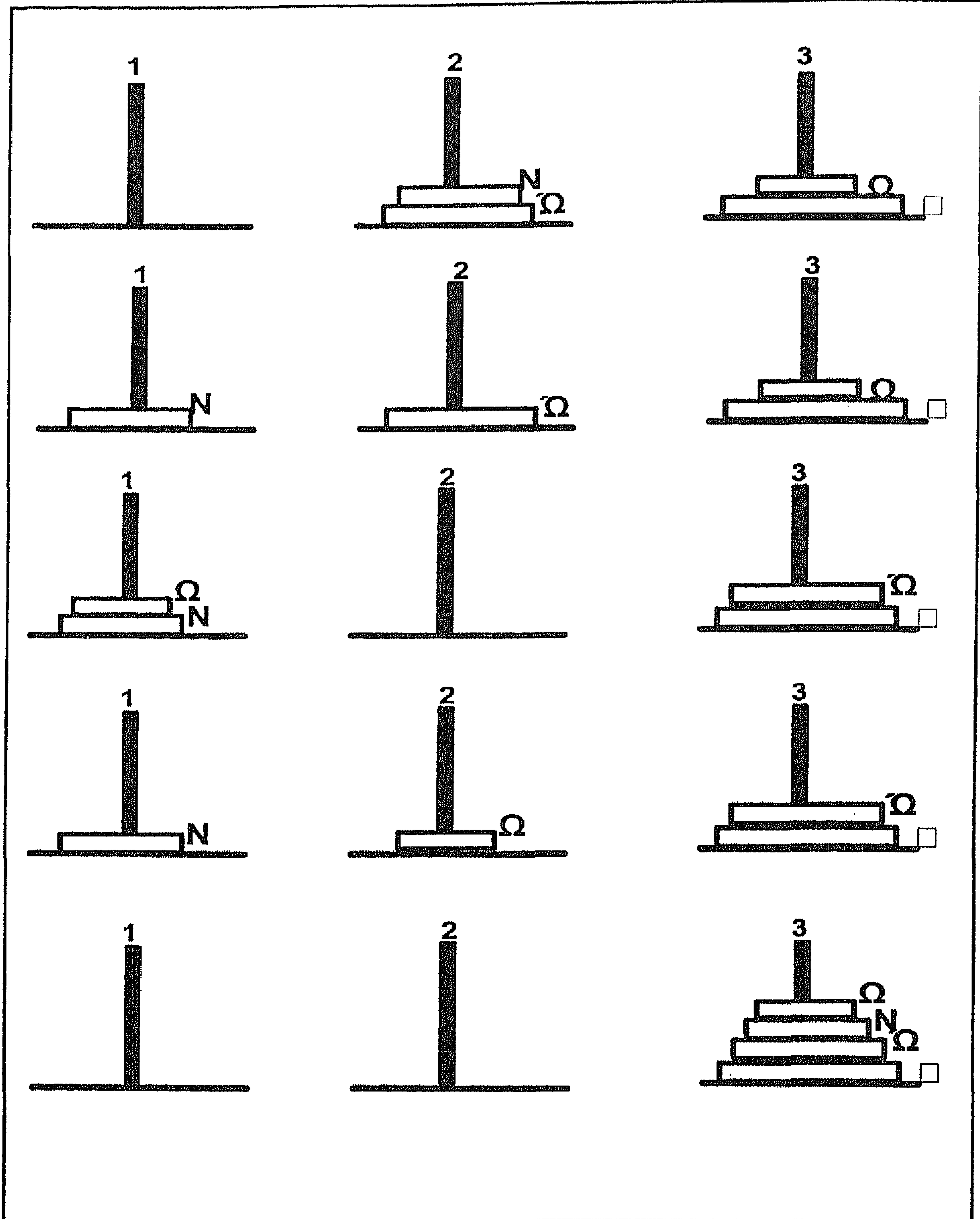
4. أبراج هانوي Towers of Hanoi

ظهرت خلال القرن التاسع عشر أحجية خاصة في أوروبا. وقد زعم المروجون لهذه الأحجية أنها كانت تمثل بالفعل صورة ما كان يجري آنذاك في معبد براهما في شرق آسيا. حيث تقول أسطورة هذه الأحجية أن الرهبان في المعبد قد أعطوا عند خلق العالم لوحة خاصة. وعلى هذه اللوحة كانت هناك ثلاث مسلات من الألماس، وعلى الأولى منها تم تثبيت أربعة وستين قرصاً من الذهب فوق بعضها، وكل قرص منها أصغر في المساحة من القرص الذي يليه من الأسفل. وقد طلب من هؤلاء الرهبان أن يقوموا بنقل هذه الأقراص بأسلوب منظم من المسلة الأولى إلى المسلة الثالثة، واحداً بعد الآخر، بالاستعانة بالمسلة الثانية شريطة أن تحتفظ هذه الأقراص بالنظام الموجود عليه وأن لا يتم نقل أكثر من قرص واحد في المرة الواحدة وأن لا يوضع أي قرص فوق قرص أصغر منه خلال مراحل النقل. ومما يدعو للطرافة أن هؤلاء الرهبان كانوا قد أخبروا بأن إنجازهم لهذه المهمة سيكون مؤشراً على نهاية العالم. وقد عرفت هذه الأحجية باسم أبراج هانوي نظراً للشكل العام الذي تتخذه (انظر الشكل (3)).



الشكل (3): أبراج هانوي (Towers of Hanoi)

وقبل أن نبيّن من الناحية البرمجيّة كيف يمكن حل هذا اللغز أو هذه الأحجية، سنحاول أولاً إيضاح الأسلوب العام للحل وعلاقة ذلك بمفهوم الاستدعاء الذاتي. ولنفترض الآن لأغراض التبسيط أن عدد الأقراص هو أربعة بدلاً من أربعة وستين قرصاً كما هو مذكور أعلاه. وعليه فإن خطوات الحل ستسير على النحو الموضح في الشكل (4).



الشكل (4): خطوات نقل ثلاثة أقراص في أحجية أبراج هانوي

وكما يتضح من هذا الشكل، فإن هذه الأحجية لا يبدو أنها تتسم بطبيعة تكرارية من النوع الذي أشرنا إليه في التطبيقين السابقين. ولكن رغم ذلك يمكن أن نجد في حلها ما يمكن أن يخضع لأساليب الاستدعاء الذاتي. فلو افترضنا أن لدينا حلاً لعدد من الأقراص مساوياً لـ: (n - 1) فإن من الممكن إيجاد حل لعدد (n) باستخدام الحل السابق لـ: (n - 1).

وإذا تيسر مثل هذا الحل المنطقي، فإن من الممكن أن نعبر عنه بأسلوب الاستدعاء الذاتي. ففي حالة المثال السابق تمكنا في لحظة معينة من نقل (n - 1) من الأقراص إلى القاعدة الثانية بمساعدة المسلة أو القاعدة الثالثة. مما أتاح لنا بالتالي نقل القرص "أ" إلى الموضع النهائي وهو القاعدة الثالثة، وذلك على النحو المبين في الشكل (5) ثم تمكنا بمساعدة القاعدة الأولى من نقل الأقراص الموجودة على القاعدة الثانية إلى مصيرها النهائي.

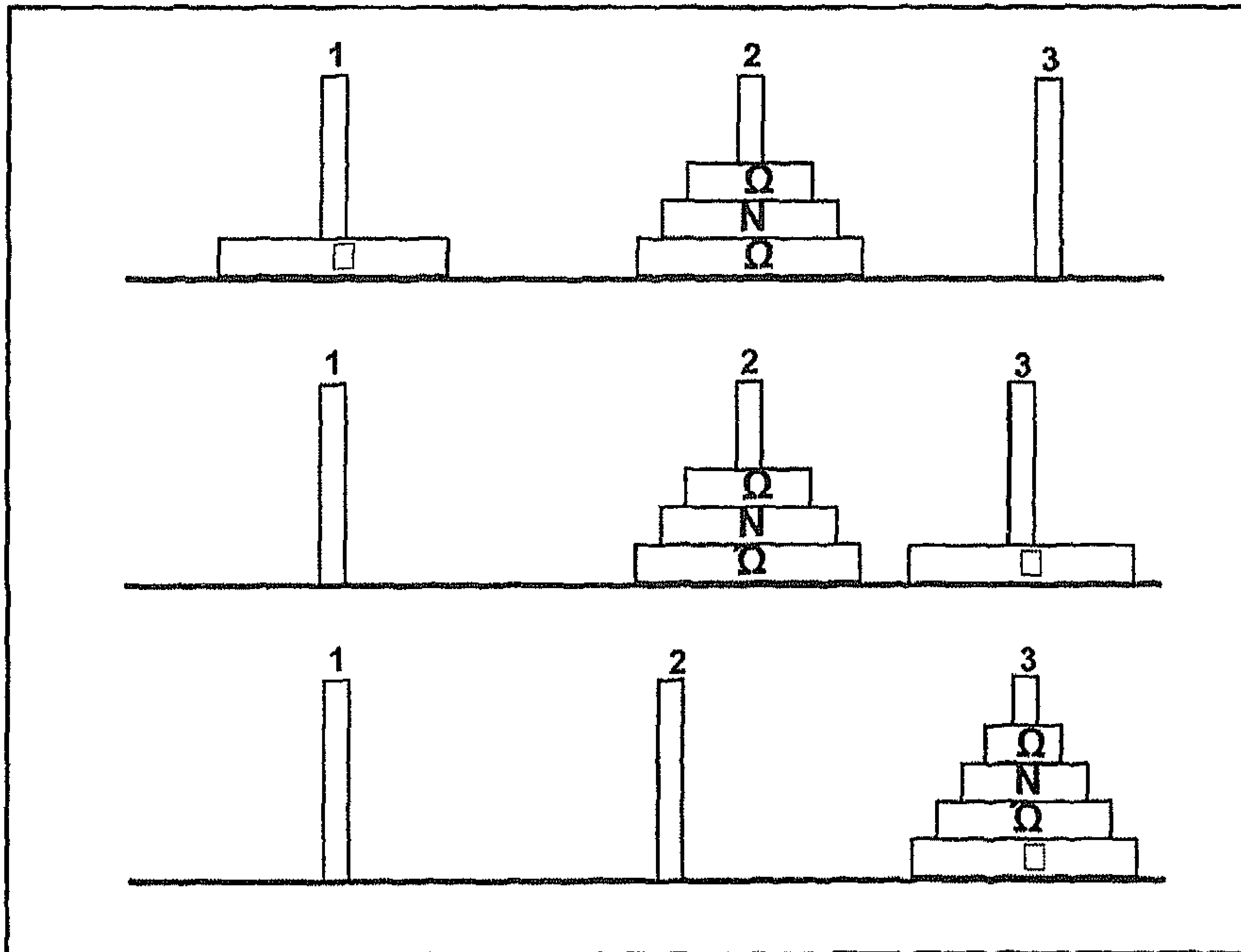
وبناء على هذا المفهوم، فإن من الممكن التعبير عن الخطوات العامة التي تم تطبيقها في الشكل (5) كما يلي:

أولاً: عندما يكون هناك قرص واحد (n=1)، انقل هذا القرص من المسلة (1) إلى المسلة (3) وتوقف.

ثانياً: انقل الأقراص الموجودة على السطح باستثناء القرص الأخير (أي: العدد n-1 من الأقراص) من المسلة (1) إلى المسلة (2) بالاستعانة بالمسلة (3).

ثالثاً: انقل القرص المتبقي على المسلة (1) إلى المسلة (3).

رابعاً: انقل الأقراص التي تم نقلها في الخطوة الثانية أعلاه وعددها (n-1) إلى مكانها النهائي وهو المسلة (3) بالاستعانة بالمسلة (1).



الشكل (5): تطبيق مفهوم الاستدعاء الذاتي على أبراج هانوي

وهذه الخطوات كفيلة بأن تقدم لنا الحل المطلوب لهـــذا اللغز. فـإذا كان عدد الأقـراص (n=1) فإن الخطوة الأولى كافية. وإذا كان عدد الأقراص (n=2)، فإن الحل القائم على نقل عدد (n-1) من الأقراص على النحو المقرر في الخطوة الثانية سيقدم لنا النتيجة المطلوبة. وبالمثل إذا كان عدد الأقراص (n=3)، فإن الحل الذي توصلنا إليه في حالة وجود قرصين فقط يمكن أن يستخدم كحل جزئي، لأنه يمثل (n-1) في هذه الحالة. وبالتالي فإن الخطوتين الثانية والرابعة ستعملان بالشكل المطلوب. وهذا يمكن أن ينطبق على الأعداد الأخرى المتوالية من الأقراص. وعليه، فإن الحلول الجزئية المتوالية يمكن أن تؤدي إلى إيجاد الحل المنطقي المطلوب لأي عدد من الأقراص. ومعنى ذلك أن بالإمكان تطبيق مفهوم الاستدعاء الذاتي على هذه المشكلة.

وإذا كانت هذه هي العلاقة القائمة بين مشكلة أبراج هانوي وأسلوب الاستدعاء الذاتي، فإن من الواضح بالتالي أن فكرة المكدسات تتصل اتصالاً جذرياً بهذه المسألة. فكما ترى، لدينا ثلاث مسلات ثابتة، وكل منها يعمل بأسلوب المكدس تماماً. وعلى هذا الأساس، فإن أي حل مطروح لهذه المشكلة، سواء باستخدام الاستدعاء أو أي أسلوب آخر، لا بد وأن يأخذ بالحسبان هذه الخاصية الأساسية لعملية نقل الأقراص من مسلة إلى أخرى. والآن لنر، عزيزي الدارس، كيف يمكن أن ننفذ الحل من الناحية البرمجية.

```
void tower(char from,char to,char thru, int n)
 {//moving N disks using the needles FROM, TO, and THRU
  if (n==1)
  cout<<" move disk 1 from needle "<<
       from<<" to needle "<<to<<"."<<endl;
   else
   {// recursive step
    tower(from, thru, to, n-1);
    cout<<" move disk "<<n<<" from needle "<<
         from<<" to needle "<<to<<"."<<endl;
    tower(from, thru, to, n-1);
   }
 }
```

وبناءً على هذه الدالة، فإن لدينا ثلاثة متغيرات تعبر عن المسلات الثلاثة ومتغير رابع هو "n" وذلك للتعبير عن عدد الأقراص التي نرغب في نقلها. وكما تلاحظ فإننا لم نعرف المتغيرات الثلاثة المعبرة عن المسلات على أنها مكدسات ولو حدث ذلك لحجزنا لها العدد الكافي من المواضع كي تستوعب القيم المتتالية التي توضع على كل مكدّس. ولو عدت للتطبيقين السابقين، لوجدت أننا فعلنا شيئاً مماثلاً. فكل متغير من هذه المتغيرات تم تعريفه على أنه ينتمي لنمط بياني بسيط وهو (char). وبحكم أسلوب تخزين هذا النمط في الذاكرة فإنه يعبر عن رمز واحد فقط. والسؤال إذن هو: أين موقع المكدسات في هذه التعريفات؟ والجواب هو ما ذكرناه في بداية هذا القسم؛ وهو أن مسألة استخدام المكدسات للتطبيقات التي تستخدم مفهوم الاستدعاء الذاتي هي مسألة مرتبطة بنظام اللغة المستعملة وغير منظورة بالنسبة للمبرمج. فالحجز يتم بعيداً عن أنظار المبرمج، ودون الحاجة إلى تعريف صريح من جانبه.

والآن لو حاولنا أن نستدعي هذه الدالة على النحو التالي:

tower('A','C','B',3); لوجدنا أن تسلسل عمليات تحريك الأقراص من مسلة إلى أخرى سيكون كما يلي:

```
move disk 1 from needle A to needle C.
move disk 2 from needle A to needle B.
move disk 1 from needle A to needle C.
move disk 3 from needle A to needle C.
move disk 1 from needle A to needle C.
move disk 2 from needle A to needle B.
move disk 1 from needle A to needle C.
```

وكما يتضح من الشكل (6 – أ) فإن الاستدعاء الخارجي المذكور أعلاه للدالة "tower" يؤدي إلى إسناد القيم المرسلة إلى المتغيرات الشكلية الأربعة. وبعد الدخول إلى الجمل الموجودة في الدالة يتم استدعاء الدالة ذاتياً للمرة الأولى، ثم يتم ذلك للمرة الثانية حتى تصبح قيمة المتغير "n" مساوية لواحد (n =1). وبذلك تتم طباعة العبارة الأولى المذكورة أعلاه. وبعد العودة تتم طباعة العبارة الثانية.

ومن المهم في هذه النقطة أن نلاحظ بأنه بمجرد العودة من السجل الثالث الموضوع على المكدس في هذا الشكل والذي رمزنا إليه بالاسم "AR3" فإن هذا السجل يحذف من المكدس ويبقى لدينا السجلان "AR2, AR1" على المكدس. وبمجرد أن نقوم بطباعة العبارة رقم (2)"...move" ونصل إلى جملة الاستدعاء الذاتي الثانية،

فإن هذا يؤدي إلى وضع جديد بالنسبة للمكدس وهو ما يوضحه الشكل (6-ب). فكما هو واضح من هذا الشكل، فقد تم حذف السجل "AR_3" وإضافة السجل "AR_4" بناء على تنفيذ جملة الاستدعاء الذاتي الثانية الواردة في الدالة والتي يتم تنفيذها لأول مرة. ونتيجة ذلك، كما ترى، هي طباعة العبارة الثالثة المذكورة أعلاه.

وبمجرد أن ننتهي من تنفيذ ما انطوى عليه السجل "AR_4" فإن التحكم بسير العمليات ينتقل إلى جملة الطباعة "move..." الواردة في السجل "AR_1"، كما يشير بذلك سهم العودة المبين في الشكل (6-ب). ولعلك تتساءل عن السبب في العودة إلى "AR_1" بدلاً من "AR_2" والإجابة هي، ببساطة، أننا انتهينا من كل ما يتضمنه "AR_2" وبناء على ذلك، يتم حذف السجلين "AR_4" و"AR_2"، تباعاً. ويبقى على المكدس السجل "AR_1" فقط. ولو تأملت الشكل (6-أ) مرة أخرى، لوجدت أن النقطة التي غادرنا منها حين كنا في السجل "AR_1" هي جملة الاستدعاء الأولى، كما يشير إلى ذلك السهم الموجود في الشكل. وبذلك فإن نقطة العودة، كما ترى، هي الجملة التالية لذلك مباشرة، وهي جملة الطباعة 'move disk n..." . وهذا يؤدي إلى طباعة العبارة الرابعة المذكورة أعلاه، والانتقال إلى جملة الاستدعاء التالية. وبذلك نصل إلى الوضع المبين فـي الشكـل (6-ج).

وكما هو واضح في الشكل (6-ج)، فإن تنفيذ جملة الاستدعاء الثانية الواردة في السجل "AR_1" أدت إلى إيجاد نسخة جديدة من المتغيرات والجمل تم تضمينها في سجل جديد تمت إضافته إلى المكدس وهو"AR5". وأول جملة يتم تنفيذها في هذا السجل الجديد هي جملة الاستدعاء الأولى. وهذا يؤدي إلى إيجاد وضع جديد وسجل جديد هو "AR_1" ونظراً لأن قيمة المتغير "n" قد وصلت إلى الحد الأدنى وهو "n=1" فإن ذلك يؤدي إلى طباعة العبارة "move disk 1..." وهي العبارة التي تحمل الرقم "5" في التسلسل المذكور أعلاه. وبعد ذلك يتحول التحكم بسير العمليـات إلى الجملـة التاليـــة مباشـــرة لجملة الاستدعاء الأولى كما يشير لذلك السهم الموجود، ويتم حذف السجل "AR_6" وهذا يؤدي إلى تنفيذ الجملة "move disk n..." وبالتالي طباعة العبارة التي تحمل الرقم "6" في السياق المذكور أعلاه. ثم يتلو ذلك تنفيذ جملة الاستدعاء الذاتي الثانية، وبذلك نصل إلى الوضع المبين في الشكل (6 - د).

	move disk 1 from needle from to needle to				return
	from=(A)	to=(C)	thru=(B)	n=1	AR_3
	tower(thru, to, from, n-1);				
	move disk n from needle from to needle to				نقطة العودة
نقطة المغادرة	tower(from, thru, to, n-1);				
	from=(A)	to=(B)	thru=(C)	n=2	AR_2
	tower(thru, to, from, n-1);				
	move disk n from needle from to needle to				
نقطة المغادرة	tower(from, thru, to, n-1);				
استدعاء خارجي	from=(A)	to=(C)	thru=(B)	n=3	AR_1

tower('A','C','B',3);

الشكل (6-أ) وضع المكدس بعد الاستدعاء الثالث

(ARi: هو سجل النشاط ويضم المتغيرات والعمليات على التوالي)

	move disk 1 from needle from to needle to				return
	from=(C)	To=(B)	thru=(A)	n=1	AR_4
نقطة المغادرة	tower(thru, to, from, n-1);				return
	move disk n from needle from to needle to				
	tower(from, thru, to, n-1);				
	from=(A)	To=(B)	thru=(C)	n=2	AR_2
	tower(thru, to, from, n-1);				
	move disk n from needle from to needle to				نقطة العودة
	tower(from, thru, to, n-1);				
	from=(A)	To=(C)	thru=(B)	n=3	AR_1

الشكل (6-ب) وضع المكدس بعد حذف "AR_3" وإضافة "AR_4"

	move disk 1 from needle from to needle to				
	from=(B)	To=(A)	thru=(C)	n=1	AR_6
	tower(thru, to, from, n-1);				
	move disk n from needle from to needle to				نقطة العودة
نقطة المغادرة	tower(from, thru, to, n-1);				
	from=(B)	to=(C)	thru=(A)	n=2	AR_5
نقطة المغادرة	tower(thru, to, from, n-1);				
	move disk n from needle from to needle to				
	tower(from, thru, to, n-1);				
	from=(A)	to=(C)	thru=(B)	n=3	AR_1

الشكل (6-ج) وضع المكدس بعد إضافة "AR_5" و"AR_6"

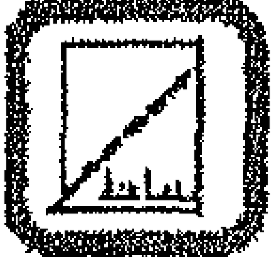
	move disk 1 from needle from to needle to				return
	from=(A)	to=(C)	thru=(B)	n=1	AR_7
نقطة المغادرة	tower(thru, to, from, n-1);				return
	move disk n from needle from to needle to				
	tower(from, thru, to, n-1);				
	from=(B)	to=(C)	thru=(A)	n=2	AR_5
	tower(thru, to, from, n-1);				return
	move disk n from needle from to needle to				
	tower(from, thru, to, n-1);				
	from=(A)	to=(C)	thru=(B)	n=3	AR_1

tower('A','C','B',3); الاستدعاء الخارجي

(الجملة التالية مباشرة (نقطة العودة..................

الشكل (6-د) وضع المكدس بعد حذف "AR_6" وإضافة "AR_7"

وكما هو واضح في الشكل (6 - د) فإن تنفيذ جملة الاستدعاء الثانية الواردة في السجل "AR_5" أدت إلى إضافة سجل جديد إلى المكدس وهو "AR_7" . وطالما أن قيمة "n" قد وصلت إلى الحد الأدنى، فإن ذلك يؤدي إلى طباعة العبارة التي تحمل الرقم "7" في التسلسل الوارد أعلاه. وبعد ذلك تتم العودة إلى نقطة الاستدعاء كما يشير السهم الموجود لذلك. ونظراً لأننا قد انتهينا من تنفيذ الجملة الواردة في السجل "AR_5"، فإننا نعود مرة أخرى إلى السجل "AR1". وكما تلاحظ، فإننا قد انتهينا أيضاً من تنفيذ جميع الجمل الواردة في هذا السجل، الأمر الذي يعني العودة إلى حيث تم استدعاء الدالة من الخارج. وهنا يتحول التحكم بسير العمليات إلى الجملة التالية مباشرة لجملة الاستدعاء، كما يتضح من تسلسل الأسهم الواردة في الشكل (6 - د). وبالنتيجة فإن جميع السجلات الواردة في هذا الشكل ستحذف من المكدس واحداً بعد الآخر حتى يصبح المكدس شاغراً عند العودة من السجل "AR1" ومغادرة الدالة والعودة إلى موقع الاستدعاء.



نشاط

بعد أن بيّنا كيف يعمل المكدس في حالة وجود ثلاثة أقراص، فإن بوسعك الآن أن تقوم بالعملية نفسها لأكثر من ذلك. والمطلوب منك هو أن تفعل هذا بالنسبة لأربعة أقراص ولعلك تذكر بأننا قمنا بنقل هذه الأقراص بطريقة يدوية من قبل ولكننا اعتمدنا في فعل ذلك على التخمين والتجربة حتى وصلنا إلى النتيجة الموضحة في الشكل (4). والآن بإمكاننا أن نقوم بالمهمة ذاتها ولكن بطريقة منهجية فهل يا ترى سنصل إلى النتيجة ذاتها وهل ستتطابق حركات النقل؟

5. الخلاصة

برامج الاستدعاء الذاتي هي برامج فرعية تستدعي نفسها. وبحل مسألة ما بكتابة برنامج فرعي يجب أن تتوفر ميزتان في المسألة هما:

1. يجب أن تتوفر حالة (أو أكثر) أساس لا يتطلب حلها استدعاءً ذاتياً.
2. عند استدعاء البرنامج الفرعي استداعاءً ذاتياً يتم ذلك على حالة أبسط (أي أقرب إلى الحالة الأساس) من الحالة الأصلية.
3. تستخدم المكدسات في عملية تنفيذ برامج الاستدعاء الذاتي وذلك لتخزين رقم الجملة التي تجب العودة إليها عند الانتهاء من تنفيذ جملة الاستدعاء إضافة إلى قيم المتغيرات والعوامل الخاصة بالبرنامج المستدعى، مثل قيم متغيرات محلية وغيرها والتي قد تستخدم عند العودة إليه.

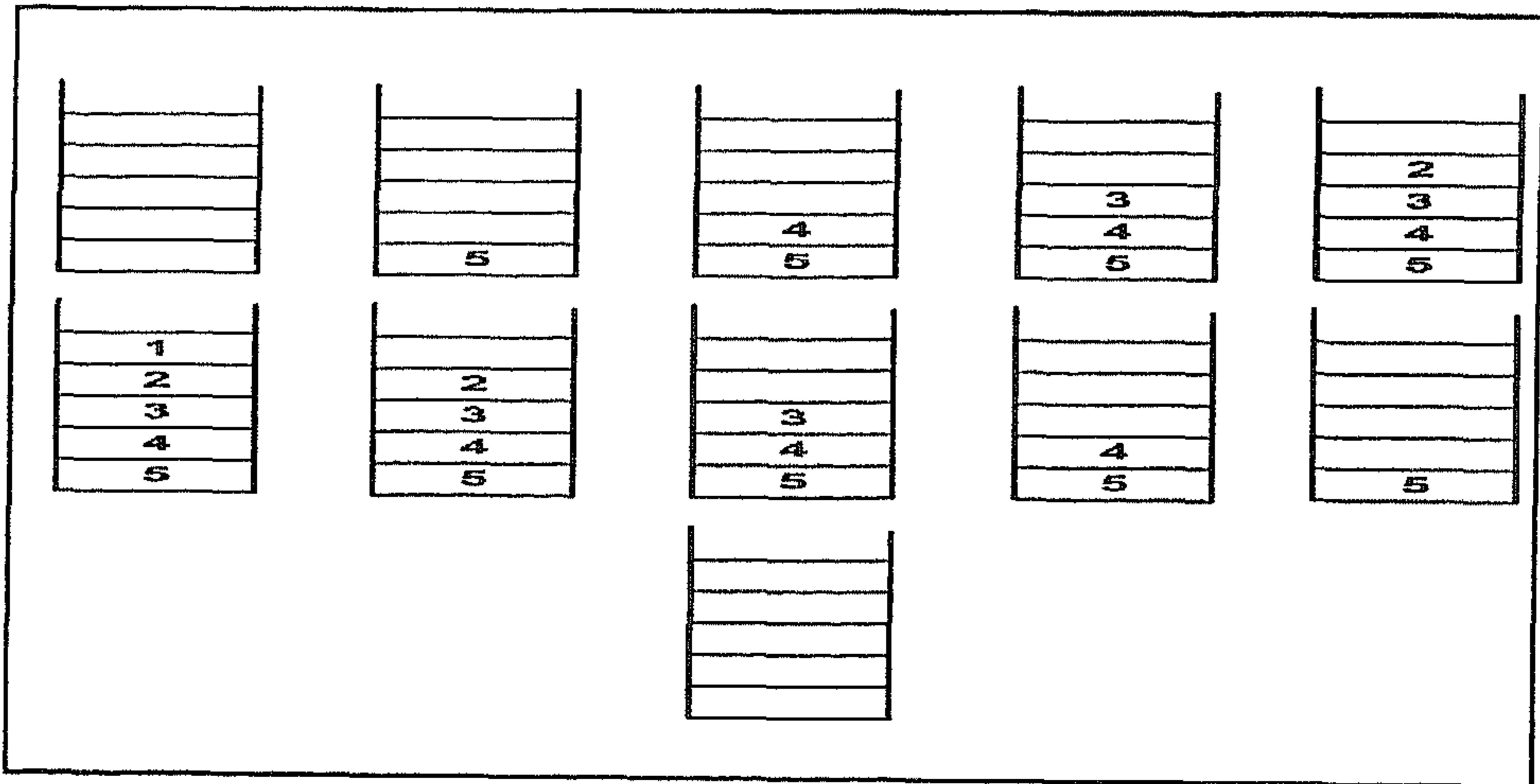
6. لمحة عن الوحدة الدراسية التاسعة

في الوحدة التالية سنتاقش، عزيزي الدارس، تركيبة بيانات مهمة تدعى الهياكل الشجرية وتستخدم لأغراض مختلفة منها تمثيل البيانات ذات العلاقة الهرمية ولأغراض البحث والفرز.

7. إجابات التدريبات

تدريب (1)

تتبع الشكل التالي من اليسار إلى اليمين:

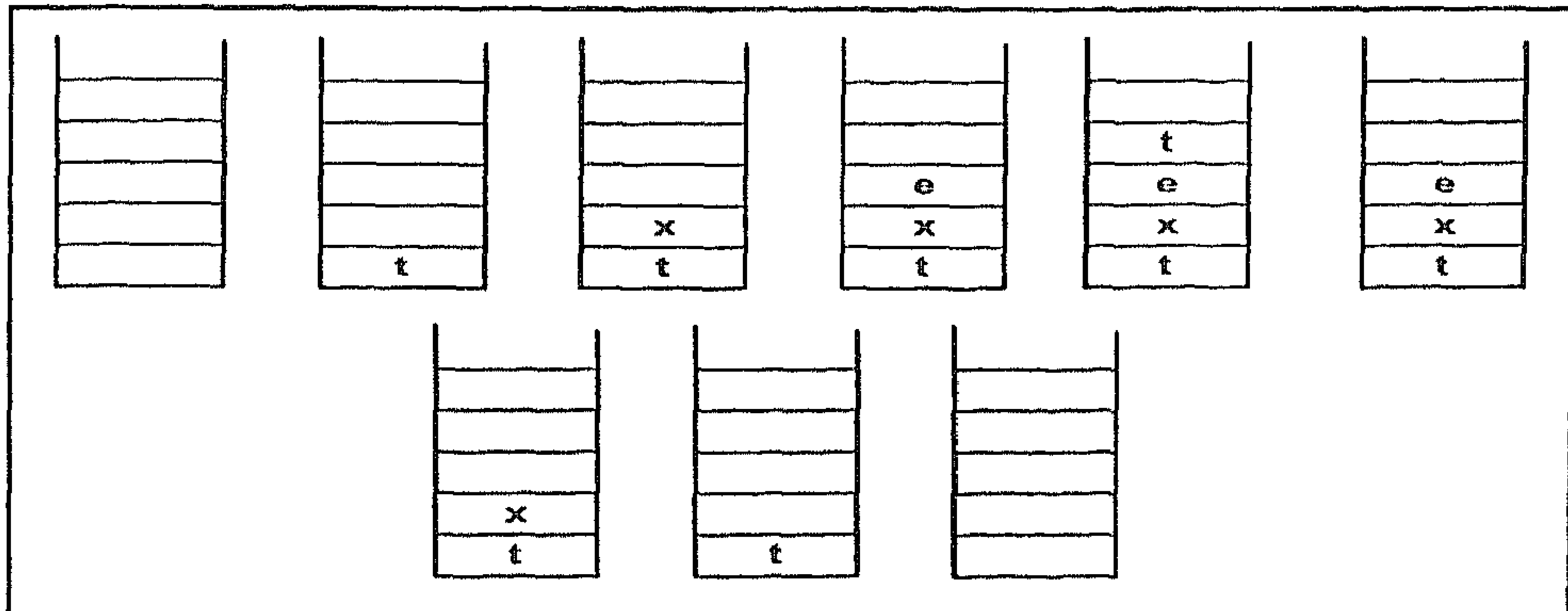


تدريب (2)

نعم وذلك لوجود جملتي استدعاء ذاتي داخل البرنامج الفرعي مما يعني أن هنالك جملتين يمكن العودة لأي منهما ولذا يجب تخزين رقم جملة العودة.

تدريب (3)

تتبع الشكل التالي من اليسار إلى اليمين:



تدريب (4)

```
void reverse2()
 { char ch;
  AStack S;
stackCreate(S);
L: if (endl)
{cin>>ch;
push(S,ch);
   goto L;
  }
  while (!stackEmpty(S))
   {pop (s,ch);
    cout<<ch;
    }
 }
```

تدريب (5)

الهدف من التدريب، عزيزي الدارس، هو جعلك تلاحظ تكلفة استخدام الاستدعاء الذاتي، إذ تتطلب البرامج التي تستخدم الاستدعاء الذاتي ذاكرة إضافية تستخدم لحفظ المكدس كذلك تتطلب هذه البرامج وقتاً تنفيذياً أكبر وهو الوقت اللازم للتخزين وللحذف من المكدس. وعليه فإننا نفضل البرنامج الذي لا يستدعي نفسه استدعاء ذاتياً ولكن المشكلة هي أنه قد يكون البرنامج الذي يستدعي نفسه ذاتياً أكثر سهولة ويسراً للكتابة والفهم، وعليه قد نضطر إلى استخدام الاستدعاء الذاتي رغم تكلفته.

8. مسرد المصطلحات

- **أبراج هانوي Towers of Hanoi**: أحجية خاصة باوروبا كانت تقول أسطورتها أن الرهبان في معبد براهما في شرق آسيا قد أعطوا عند خلق العالم لوحة خاصة، وعلى هذه اللوحة كانت هناك ثلاث مسلات من الألماس وعلى المسلّة الأولى منها 64 قرص من الذهب قد طلب من الرهبان نقلها باسلوب خاص.
- **الاستدعاء الذاتي Recursion**: يعني أن تعمل الدالة على استدعاء ذاتها بعدد محدد من المرات يتم بعدها إيجاد ناتج الدالة.
- **عنوان جملة العودة Return Address**: أي رقم الجملة التي يتوجب على البرنامج المستدعي العودة إلى تنفيذها بعد أن ينتهي من تنفيذ البرنامج المستدعى.



9. المراجع

1. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice- Hall, 2000.
2. Tremplay, J.P.; and Sorenson, P.G.; An Introduction to Data Structures with Applications, 2nd Edition., McGraw-Hill, 1984.
3. Amsbury, Wagne, Data Structures from Arrays to Priority Queues. Belmont (USA): Wadsworth, 1985.
4. Kruse, Robert L., Data Structures and Program Design. Englewood Cliffs (USA): Prentice-Hall, 1984.
5. Lipschutz, Seymour, Theory and Problems of Data Structures. New York: McGraw-Hill, 1986.
6. Miller, Lawrence H., Advanced Programming Design and Structures. New York: McGraw-Hill, 1986.
7. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison- Wesley, 1999.
8. Lewis, T. G.; Smith, M.Z., Applying Data Structures. Atlanta (USA): Houghton Mifflin, 1976.
9. Tenenbaum, Aarom M.; Augenstein, Moshe J., Data Structures Using Pascal. Englewood Cliffs (USA): Prentice-Hall, 1981.

الوحدة التاسعة

الهياكل الشجرية (Trees)

محتويات الوحدة

1. المقدمة

1.1 التمهيد

أهلا بك، عزيزي الدارس، إلى الوحدة التاسعة من كتاب "تركيب البيانات وتصميم الخوارزميات" وهي بعنوان "الهياكل الشجرية".

في هذه الوحدة، سنناقش تراكيب بيانات غير خطية لأول مرة في هذا المقرر وهي الهياكل الشجرية بأنواعها المختلفة. وهو موضوع مهم لكثرة استخدامات هذا النوع من التراكيب البيانية لأغراض مختلفة حيث تستخدم الهياكل الشجرية لأغراض تمثيل البيانات ذات العلاقة الهرمية في البحث والفهرسة والفرز وفي العديد من التطبيقات المختلفة.

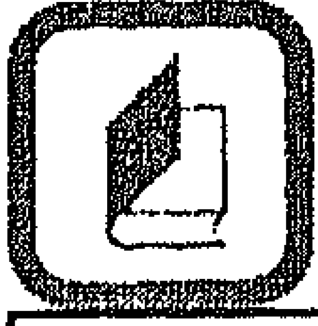
2.1 أهداف الوحدة

ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1. تُعرِّف مفهوم الهياكل الشجرية وتميز بين الأنواع المختلفة منها.
2. تمثل الهياكل الشجرية الثنائية بعدة أساليب.
3. توضح أهم العمليات المستخدمة على الهياكل الشجرية الثنائية وتعرف كيف تنفذها.
4. تعدد بعض تطبيقات الهياكل الشجرية.
5. توضح مفهوم الهياكل الشجرية من نوع B وتعرف استخداماتها.

3.1 أقسام الوحدة

تتكون هذه الوحدة من ستة أقسام رئيسة ترتبط بقائمة الأهداف السابقة. في القسم الأول ستتعرف على مفهوم الهياكل الشجرية والمصطلحات الخاصة بها ويرتبط هذا القسم بتحقيق الهدف الأول. أما في القسم الثاني فسنقدم لك الهياكل الشجرية الثنائية وطرق تمثيلها، ويرتبط هذا القسم بتحقيق الهدف الثاني. أما في القسم الثالث فسنتعرف على أهم العمليات التي تنفذ على الهياكل الشجرية الثنائية والبرامج اللازمة لتنفيذ هذه العمليات ويحقق هذا القسم الهدف الثالث. وفي القسم الرابع ستتعرف على الأنواع المختلفة للهياكل الشجرية الثنائية وعلاقة ذلك بكفاءة العمليات الواردة في القسم الثالث، ويهدف هذا القسم إلى تحقيق الهدف الأول أيضاً. ويناقش القسم الخامس التطبيقات المختلفة للهياكل الشجرية ويحقق بذلك الهدف الرابع. أما القسم السادس، وهو الأخير، فيناقش الهياكل الشجرية من نوع B ويوضح استخداماتها، ويحقق بذلك الهدف الخامس.



4.1 القراءات المساعدة

على الرغم من شمولية الأقسام والمسائل التي تم عرضها في هذه الوحدة إلا أن هناك فوائد كثيرة يمكن أن يجنيها الدارس من الرجوع إلى بعض القراءات الإضافية المساعدة. فهناك المزيد من المعلومات والأمثلة في هذه المصادر، وهناك أيضاً المزيد من التدريبات والأسئلة. ونوصي بالمصدرين التاليين لهذه الغاية، مع التذكير بأن قائمة المراجع في نهاية هذه الوحدة تتضمن مصادر على قدر كبير من الأهمية والفائدة:

1. Amsbury, Wayne, Data Structures from Arrays to Priority Queues. Belmont (USA): Wadsworth, 1985. pp. 97-169 & ,141-120 ,119-198
2. Malik, D.S. Data Structures Using C++, 1st Edition, Course Technology, Inc., 2003.
3. Main, Michael, Data Structures & Other Objects Using C++, 3d Edition, Addison-Wesley, 2004.

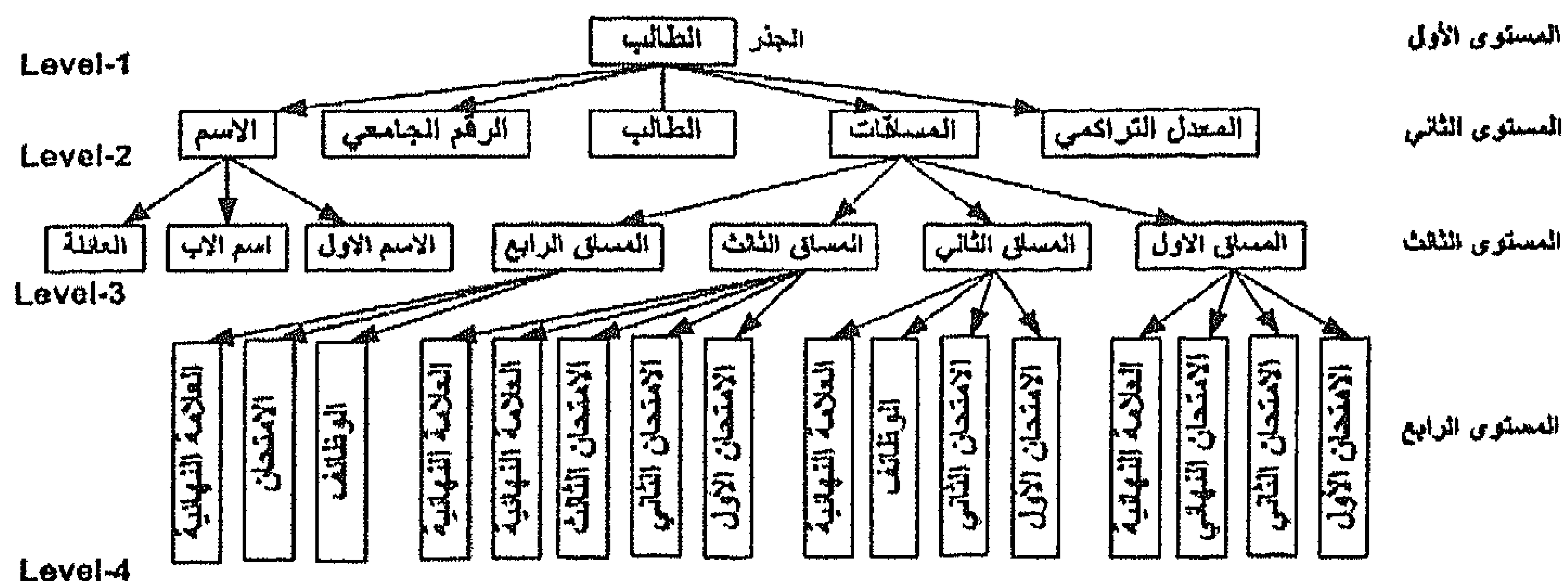
2. مقدمة إلى الهياكل الشجرية

تعرضنا في الوحدات السابقة من هذا الكتاب، عزيزي الدارس، إلى أنواع مختلفة من تراكيب البيانات مثل القوائم والجداول والطوابير والمكدسات، وتدرج هذه الأنواع من تراكيب البيانات تحت اسم التراكيب الخطية Linear Structures. وفي هذه الوحدة نعالج نوعاً آخر من تراكيب البيانات يدعى "الهياكل الشجرية"، والتي تعتبر من التراكيب غير الخطية (NonLinear Structures). وتستعمل مثل هذه التراكيب لتمثيل البيانات التي ترتبط مع بعضها البعض بشكل هرمي كما هو الحال في شجيرات أنساب العائلات والسجلات. فالسجل الواحد يحتوي على مجموعة من الحقول، ويمكن لبعض هذه الحقول أن تكون سجلات بحد ذاتها وهكذا.

ويتكون الهيكل الشجري من عناصر (Nodes) كما هو مبين في الشكل (1)، ويحتوي كل عنصر من العناصر على جزئين رئيسين، يحتوي الجزء الأول منها على مجموعة من البيانات التي تصف كياناً معيناً أو أكثر. ويعتبر هذا الجزء بمثابة الجزء الرئيس الذي نسعى للوصول إليه ومعرفة محتوياته لإجراء العمليات المناسبة عليها.

أما الجزء الثاني فيحتوي على بعض المعلومات التي تحدد علاقة العناصر ببعضها، وتعتبر هذه المعلومات ضرورية لربط عناصر الهيكل الشجري ببعضها. وبتتبع هذه المعلومات (معلومات الربط بين العناصر) يمكننا الوصول إلى عناصر الهيكل الشجري المختلفة ومعرفة البيانات المخزنة فيها. ويعتمد نوع معلومات الربط على طريقة تمثيل الهيكل الشجري داخل ذاكرة الحاسوب. وسنتطرق إلى هذا الموضوع، عزيزي الدارس، في الجزء التالي من هذه الوحدة.

ومن الشائع تمثيل العلاقة الهرمية بين عناصر الهيكل الشجري باستخدام مؤشرات الربط كما هو موضح في الشكل (1)، ومن الدارج أيضاً استخدام دوائر صغيرة لتمثيل عناصر الهيكل الشجري وبداخلها البيانات ووصل هذه العناصر مع بعضها البعض عن طريق مؤشرات الربط. وتعد الهياكل الشجرية من الأنواع الأكثر مرونة وقوة ويمكن استخدامها في مجالات عدة.



الشكل (1): الهيكل الشجري لسجل الطالب

وسنركز في هذه الوحدة على نوع خاص ومهم من الهياكل الشجرية يسمى الهياكل الشجرية الثنائية وذلك لسهولة تمثيلها ومعالجتها بواسطة الحاسوب ولكثرة استخداماتها.

ويمكن تعريف الهياكل الشجرية (Tree structures) على أنها تركيب هرمي لمجموعات من البيانات بحيث يكون لأعلى مستوى في التسلسل مجموعة واحدة تدعى الجذر(Root)، أما بقية المجموعات فينبثق كل منها من مجموعة أخـرى ذات مستوى أعلى منها، وترتبط به باستخدام مؤشرات الـربط (Pointers). ويطلق على كل مجموعة من هذه البيانات وما يلزمها من مؤشرات الربط لفظ عنصر (Node).

مع أن هذا التعريف للهيكل الشجري العام (General Tree) لا يحدد عدد مؤشرات الربط التي يمكن للعنصر الواحد أن يحتوي عليها إلا أن الكثير من التطبيقات لا تحتاج لأكثر من مؤشرين اثنين في كل عنصر لربط هذا العنصر مع العناصر الأخرى. ويدعى هذا النوع من الهياكل الشجرية بالهياكل الشجرية الثنائية، وهي الأكثر شيوعاً.

ولتحديد العلاقة بين العناصر المختلفة في الهيكل الشجري، غالباً ما نلجأ إلى استخدام المصطلحات نفسها لتحديد العلاقه بين أفراد العائلة في شجيرات الأنساب والسلالات وسنرجع عند تعريف المصطلحات إلى الشكـل (1) الذي يمثل بعض المعلومات اللازمة عن سجل الطالب المسجل في التعليم المفتوح وبشكل هيكل شجري. ومن بين هذه المعلومات:

1. اسم الطالب والذي يتكون من الاسم الأول واسم الأب واسم العائلة.

2. الرقم الجامعي للطالب.

3. عنوان الطالب.

4. المقررات التي يدرسها الطالب وتشمل المقرر الأول والمقرر الثاني والمقرر الثالث والمقرر الرابع.

5. العلامات لكل مقرر.

6. المعدل التراكمي للطالب.

وكما هو واضح في الشكل (1)؛ يقسم الهيكل الشجري إلى مستويات (Levels) تبدأ بالمستوى (1) لأول عنصر في الهيكل الشجري، ويسمى بجذر (Root) الهيكل الشجري ومن ثم يزيد كل مستوى بمقدار واحد عن المستوى الذي يسبقه.

ونلاحظ أن جذر الهيكل الشجري يبرز في الأعلى على العكس من الشجرة الحقيقية والتي يكون جذرها في الأسفل.

يطلق على العناصر التي تحتوي على البيانات: الرقم الجامعي، الاسم، والعنوان، والمقرر والمعدل التراكمي اسم أبناء الجذر (Children)، لأنها تنحدر مباشرة منه وترتبط به بواسطة مؤشرات الربط.. ويطلق على الجذر اسم المنشأ أو الأهل أو الأب/الوالد (Parent) لهؤلاء الأبناء. ونلاحظ من الشكل (1) أن مستوى الأهل يقل بمقدار واحد عن مستوى الأبناء، أي مستوى الأبناء يساوي مستوى الأهل + 1.

يطلق على مؤشر الربط بين الأهل والابن اسم الحافة (Edge) ويطلق على مجموعة متتابعة من الحواف اسم الممر(Path)، أما العنصر الذي لا ينحدر منه أي من الأبناء فيطلق عليه اسم الورقي (Leaf Node)، وقد جاءت هذه التسميه لأن ورقة الشجرة العادية لا ينبثق منها أي شيء آخر. والاسم الدارج الآخر لمثل هذا العنصر هو العنصر النهائي(Terminal Node). أما الممر الذي يبدأ من عنصر ما وينتهي بعنصر ورقي فيسمى غصن (Branch).

ويستعمل مصطلح عمق الهيكل الشجري أو ارتفاعه ليعني أكبر عدد من العناصر الموجودة في أغصان الشجرة. وهذا يعني أن عمق الهيكل الشجري يساوي أعلى مستوى في الشجرة.

بالرجوع إلى الشكل (1) نجد أن الطالب يمثل جذر الهيكل الشجري ويمثل المستوى رقم (1)، وأن جميع العناصر على المستوى (2) تنحدر مباشرة من الطالب لذلك فهي أبناء الطالب، وأن الطالب هو أهل (Parent) لهذه العناصر. أما الخطوط التي تربط الطالب مع أبنائه فتسمى الحواف (Edges) وتوجد خمس حواف تربط الطالب مع أبنائه، وأن المجموع الكلي للحواف هو (28).

ونلاحظ أيضاً أنه من الممكن أن ينحدر من عنصر الأهل عدة أبناء بينما يوجد لكل عنصر واحد والد (أهل) واحد فقط.

من العناصر النهائية في الشكل نورد الرقم الجامعي، العنوان، المعدل التراكمي، الاسم الأول، اسم الأب، اسم العائلة، الامتحان الأول للمقرر الأول... إلخ.

أما بالنسبة لعمق الشجرة فهو يساوي 4 وهو أعلى مستوى في هذا الهيكل الشجري.

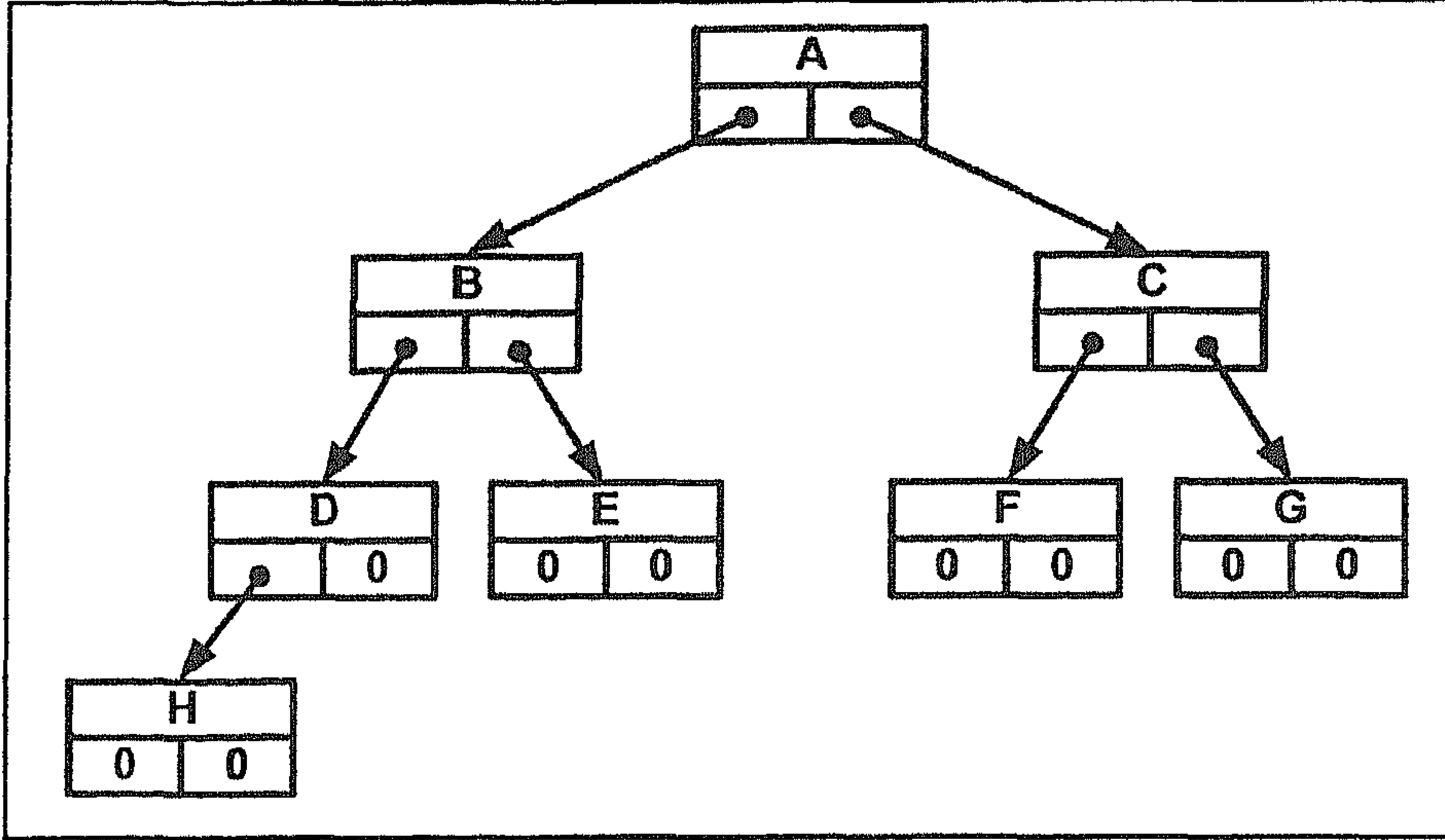
يطلق الاسم شجيرة فرعية (Subtree) على أي جزء من الهيكل الشجري، فعلى سبيل المثال، توجد خمس شجيرات فرعية منبثقة من الطالب كما يلي:

1. الشجيرة الأولى تتكون من عنصر واحد هو الرقم الجامعي.
2. الشجيرة الثانية تتكون من أربعة عناصر هي الاسم، (ويشكل جذر هذه الشجيرة) والأبناء المنبثقين عنه هم: الاسم الأول، اسم الأب، العائلة.
3. الشجيرة الثالثة وتحتوي على عنصر واحد هو العنوان، ويشكل جذر هذه الشجيرة.
4. الشجيرة الرابعة وتتكون من عنصر المقررات وما ينحدر منه، حيث ينحدر من هذا العنصر شجيرات أخرى كما هو مبين في الشكل. فعلى سبيل المثال، المقرر الأول وأبناؤه تشكل شجيرة فرعية.
5. الشجيرة الخامسة وتتكون من عنصر واحد هو المعدل التراكمي ويمثل جذر هذه الشجرة.

تجدر الملاحظة بأن المواقع المنحدرة من الوالد (الأهل) يمكن اعتبارها على أنها مرتبة من اليسار إلى اليمين. فلو أخذنا بعين الاعتبار العنصر الذي يحتوي على المقرر الأول، لوجدنا أن هناك أربعة أبناء منحدرون منه، يطلق اسم الابن الأول على الامتحان الأول، والابن الثاني على الامتحان الثاني، والابن الثالث على الامتحان الثالث، والابن الرابع على العلامة النهائية.

3 الهياكل الشجرية الثنائية Binary Trees

تتميز الهياكل الشجرية الثنائية بوجود مؤشري ربط على الأكثر في أي عنصر من عناصر الهيكل الشجري حيث أنه يمكن للعنصر الواحد أن يحتوي على حافة واحدة أو اثنتين أو أن يخلو من الحواف. كما هو واضح في الشكل (2) .



الشكل (2): هيكل شجرة ثنائية

ففي هذا الشكل نجد أن كل عنصر من العناصر التي تحتوي على البيانات A, B, C ينحدر منه ابنان اثنان (أي أن كل منهما يحتوي على مؤشري ربط لربطه مع أبنائه الاثنين)، أما العنصر الذي يحتوي على الحرف D فينبثق منه ولد واحد وأما قيمة المؤشر الثاني في هذا العنصر فهي 0 ويطلق على هذه القيمة الاصطلاح NULL في لغة سي++. وأما قيم مؤشرات الربط في العناصر التي تحتوي على القيم E ,F,G,H فجميعها 0 أو NULL .

وكما هو الحال في الهياكل الشجرية العادية، يعتبر الجذر هو نقطة الدخول إلى الهيكل الشجري حيث أنه بتتبع مؤشرات الربط يمكن الوصول إلى العناصر المختلفة، فالمؤشر الأيسر من العنصر يؤدي إلى الشجيرة الفرعية اليسرى، والمؤشر الأيمن يؤدي إلى الشجيرة الفرعية اليمنى. وبالرجوع إلى الشكل (2) نجد أن العنصر الذي يحتوي على الرمز B يمثل الابن الأيسر للجذر أو التابع الأيسر الفوري (Left Successor) للجذر، وأن العنصر الذي يحتوي على الرمز C يمثل الابن الأيمن أو التابع الأيمن الفوري (Right Successor) للجذر.

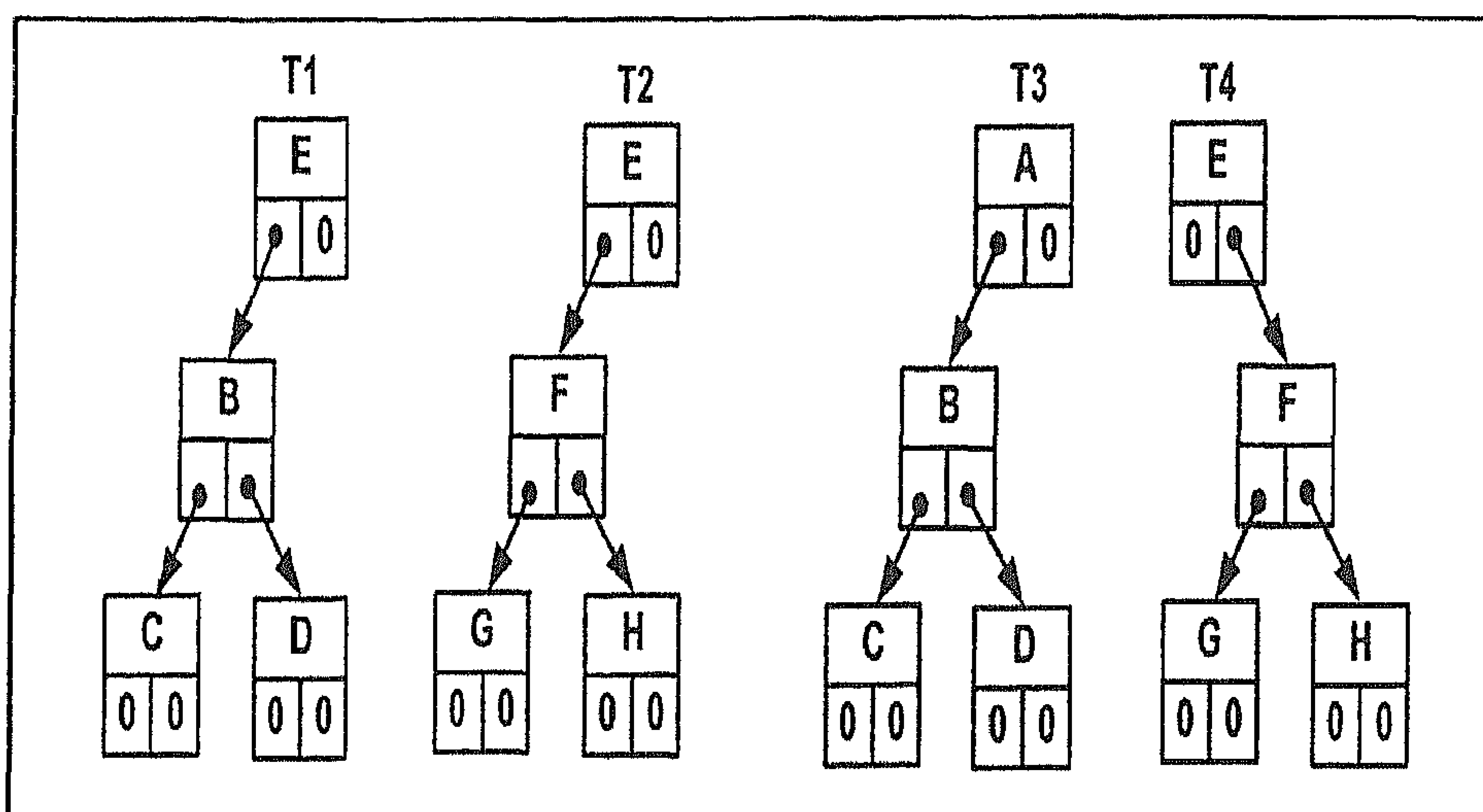
وكذلك فإن العناصر التي تحتوي على البيانات H,E,D,B تمثل الشجيرة اليسرى (Left Subtree) من هذا الهيكل الشجري، وأن العناصر التي تحتوي على البيانات C,F,G تمثل الشجيرة اليمنى (Right Subtree) وتعتبر كل شجيرة من هذه الشجيرات هيكلاً شجرياً بحد ذاتها، وينطبق عليها تعريف الهيكل الشجري.

يوجد في هذا الهيكل الشجري أربعة عناصر نهائية (Terminal Nodes) وهي العناصر التي تحتوي على البيانات H,G,F,E.

وتجدر الملاحظة أنه لا توجد عناصر مشتركة بين الشجيرة اليسرى والشجيرة اليمنى حيث أن هاتين الشجيرتين منفصلتين عن بعضهما تماماً. كذلك لا يجوز أن يؤشر أي من مؤشرات الربط إلى العناصر السابقة في الهيكل الشجري. بل إن هذه المؤشرات يجب أن تؤشر على عناصر لاحقة في الهيكل الشجري (أي أن مؤشرات الربط تربط الوالد مع أبنائه فقط).

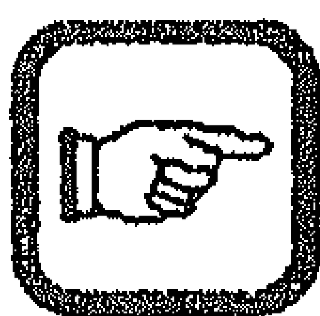
يمكننا الآن تعريف الهيكل الشجري الثنائي على أنه مجموعة من العناصر التي من الممكن أن تكون خالية (Empty)، أو تحتوي على عنصر جذري (Root Node) وشجيرتين منفصلتين عن بعضهما (Two Disjoint Binary Trees) يطلق عليهما الشجيرة اليسرى (Left SubTree) والشجيرة اليمنى (Right (SubTree. ويعتبر هذا التعريف دورياً ومتكرراً (Recursive)، أي أننا لتعريف الهيكل الشجري الثاني استخدمنا مفهوم الشجيرة اليسرى والشجيرة اليمنى وهما بحد ذاتهما هياكل شجرية ثنائية. وتعتبر هذه الخاصية للهياكل الشجرية الثنائية ميزة مهمة تسهل برمجتها كما سنرى عند بحث العمليات على الهياكل الشجرية الثنائية. إذ أن برمجة معظم هذه العمليات تتطلب أن يستدعي البرنامج نفسه استدعاءً ذاتياً.

يعتبر الهيكل الشجري الثنائي T1مشابهاً للهيكل الشجري الثنائي T2 إذا كان لهما التركيب نفسه (أي الشكل نفسه)؛ ففي الشكل (3) يُعتبر T1 مشابهاً لـ T2 لتشابههما في التركيب، مع أن البيانات الموجودة في العناصر المتناظرة مختلفة. تعتبر هاتان الشجيرتان مشابهتين للشجرة T3 أيضاً. وفي حالة كون الشجيرتين متشابهتين وعندما تحوي العناصر البيانات نفسها في المواقع المتناظرة، يقال عن هذه الشجيرات بأنها نسخ طبق الأصل (Copies). ففي الشكل (3) تعتبر T1 نسخة طبق الأصل لـ T3 . أما الشجيرة الثنائية T4 فهي غير مشابهة ولا نسخة طبق الأصل لأي من الشجيرات الأخرى في الشكل.



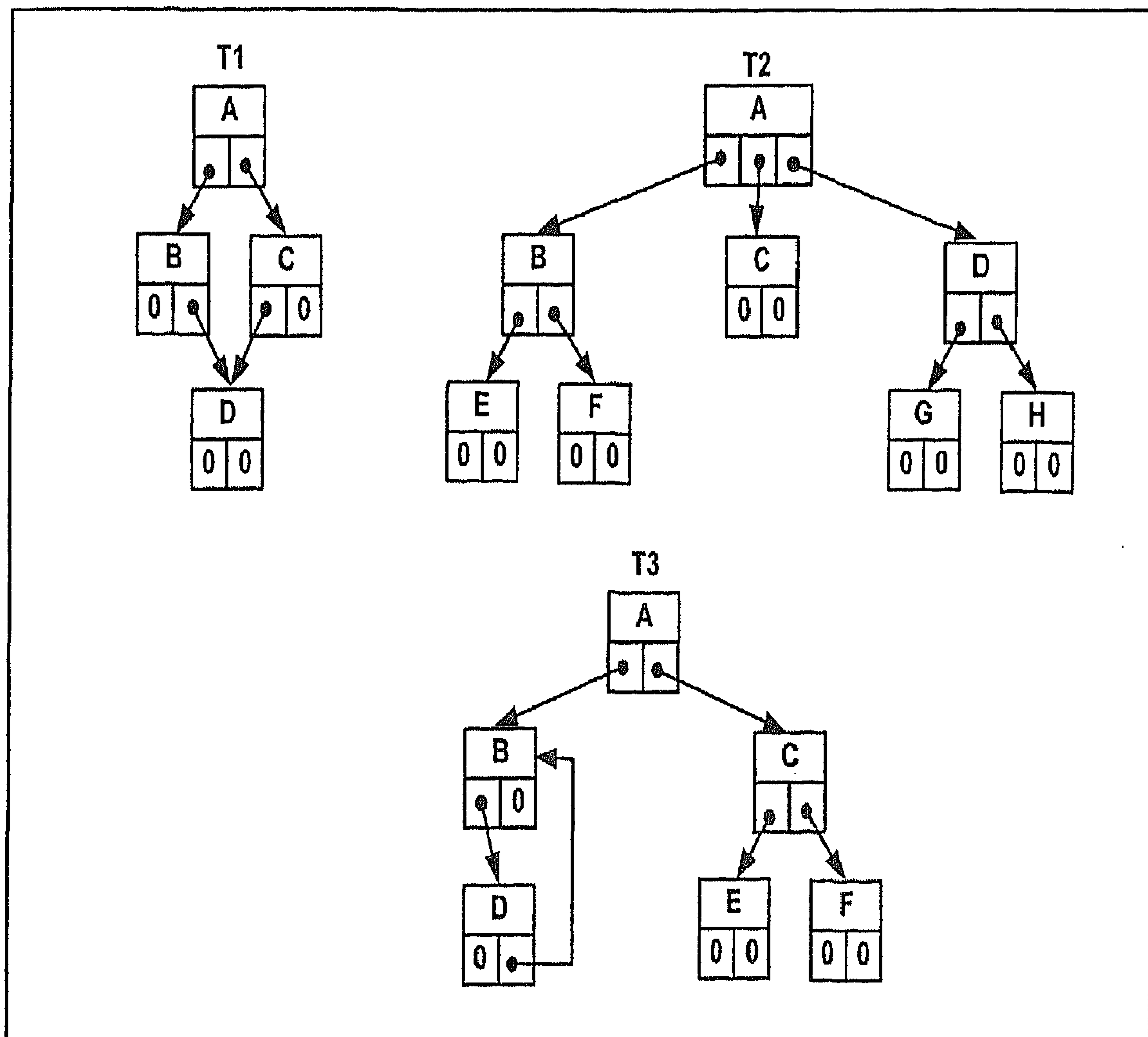
الشكل (3) أمثلة على الهياكل الشجرية الثنائية

من هنا يجب التمييز، عزيزي الدارس، بين التابع الأيسر (Left Successor) والتابع الأيمن (Right Successor) للهيكل الشجري الثنائي. ومن هذا المنطلق يجب التمييز بين الشجيرة T2 والشجيرة الثنائية T4 في الشكل (3)، ذلك أن العنصر F في الشجيرة الثنائية T2 الابن الأيسر لعنصر الجذر E، بينما يمثل العنصر F في الشجيرة الثنائية T4 الابن الأيمن لعنصر الجذر E.



مثال (1)

بين أسباب عدم صحة الهياكل الشجرية الثنائية التالية (الشكل (4)):



الشكل (4)

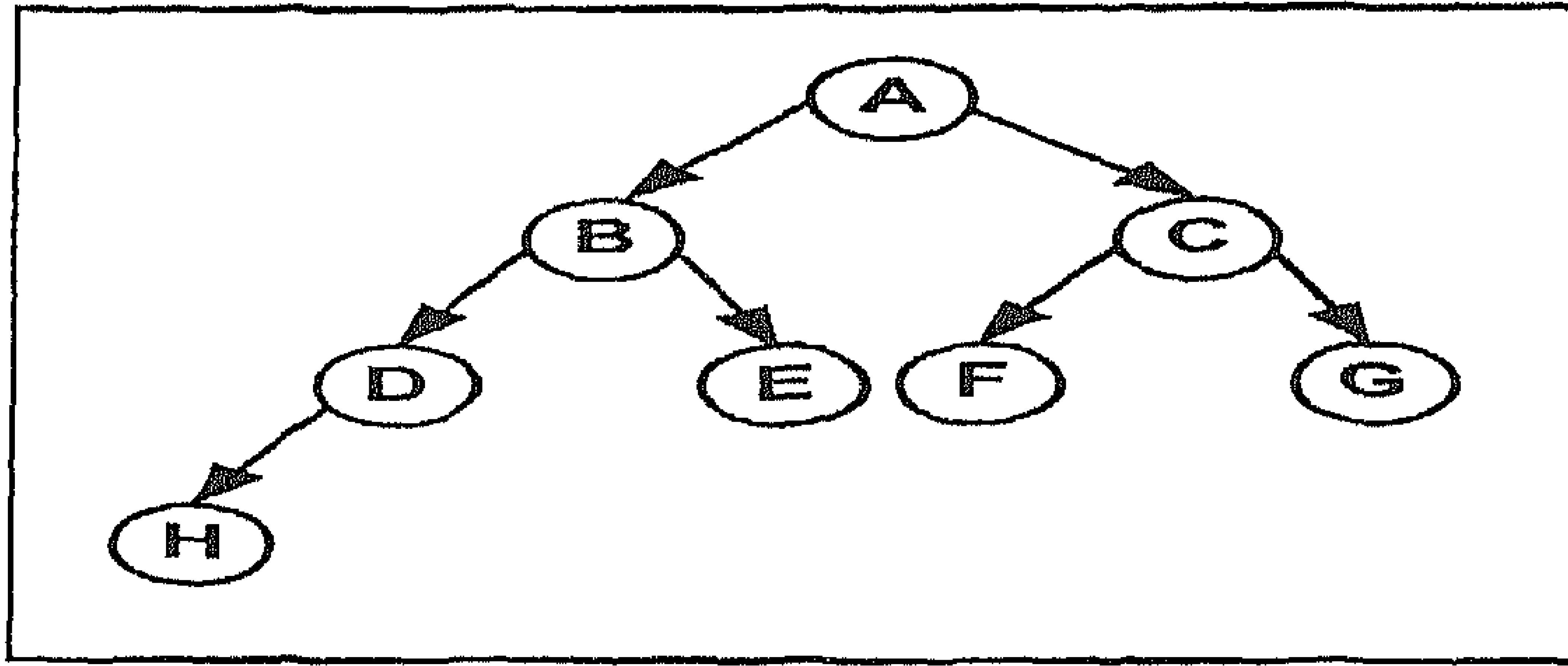
الحل:

1. بالنسبة للهيكل الشجري الثنائي T1 فسبب عدم صحته أن العنصر الذي يحتوي على الرمز D ينبثق من العنصرين B وC، أي أن لهذا العنصر والدين وهذا لا يجوز.
2. أما بالنسبة للشكل T2 فهو غير صحيح وذلك لأنه ينبثق من الجذر ثلاثة مؤشرات ربط، وهذا غير جائز لأنه يجب أن لا يزيد عدد مؤشرات الربط من أي عنصر عن اثنين في الهياكل الشجرية الثنائية.

3. أما بالنسبة للشكل T3 فهو لا يمثل هيكلاً شجرياً ثنائياً لأن المؤشر الأيمن من العنصر الذي يحتوي على الرمز D، يشير إلى الوالد (أي العنصر الذي يحتوي على الرمز (B) وهذا غير جائز.

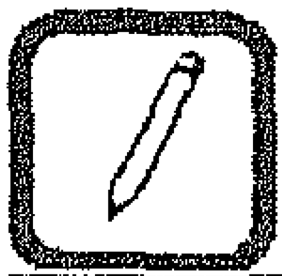
من الآن فصاعداً سأحاول استخدام الرموز للإشارة إلى العنصر، ولذلك سأحاول وضع هذه الرموز لوحدها عوضاً عن العنصر، وسأشير إلى مؤشرات الربط باستخدام خطوط مستقيمة.

فعلى سبيل المثال يمكن إعادة رسم الهيكل الشجري الوارد في الشكل (2) ليصبح كما يلي:



الشكل (5)

حيث تشير A إلى جذر الهيكل الشجري وتستخدم B للإشارة إلى الابن الأيسر للجذر، وتستخدم C للإشارة إلى الابن الأيمن للجذر، وهكذا.



تدريب (1)

إشارة إلى الهيكل الشجري الثنائي التالي، أجب على الأسئلة التالية:

أ- ما جذر هذه الشجرة ؟

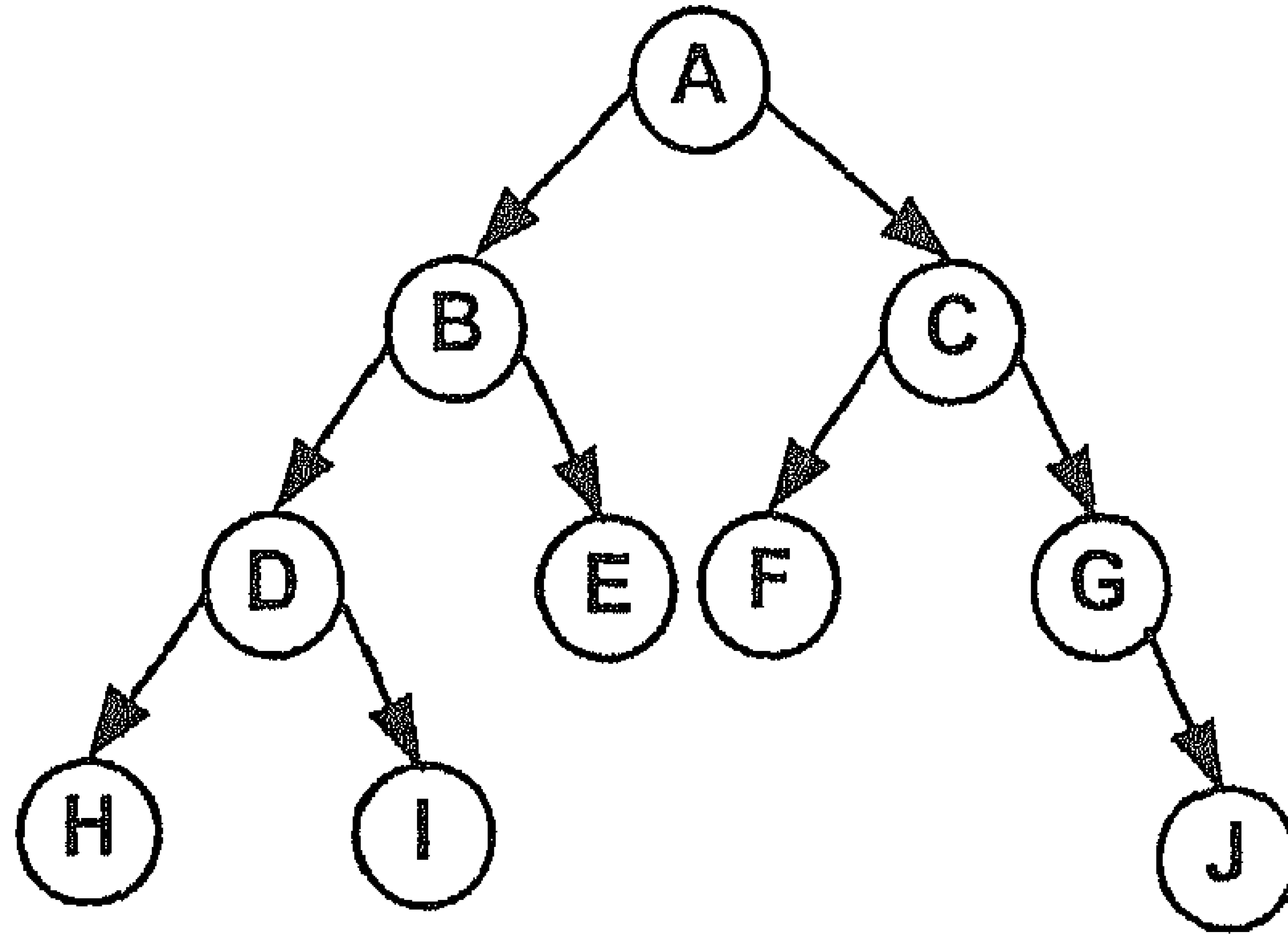
ب- من هم أبناء العنصر B ؟

ج- ما العناصر الناتجة من العنصر (خَلَفْ) B ؟

د- ما والد العنصر B ؟

هـ- ما العناصر الورقية ؟

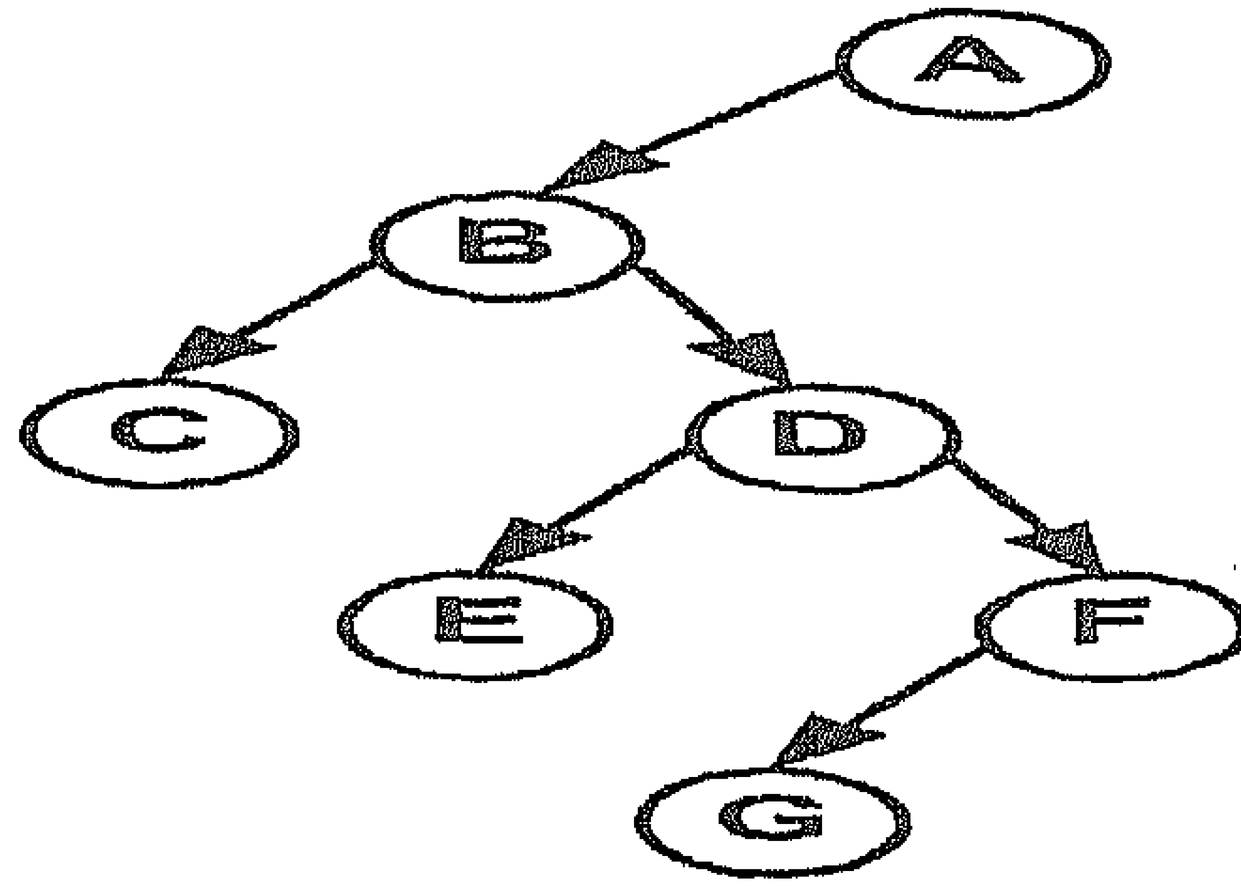
و- ما ارتفاع (عمق) هذه الشجرة ؟



الشكل (6)

أسئلة التقويم الذاتي (1)

أجب على الأسئلة التالية بالرجوع إلى الهيكل الشجري التالي:



الشكل (7)

1. ما جذر الشجيرة اليمنى للعنصر B ؟
2. ما العناصر الطرفية (الورقية) ؟
3. ما العناصر غير الورقية ؟
4. ما مستوى كل من العناصر المكونة للهيكل الشجري ؟
5. ما عمق هذا الهيكل الشجري ؟

طرق تمثيل الهياكل الشجرية الثنائية

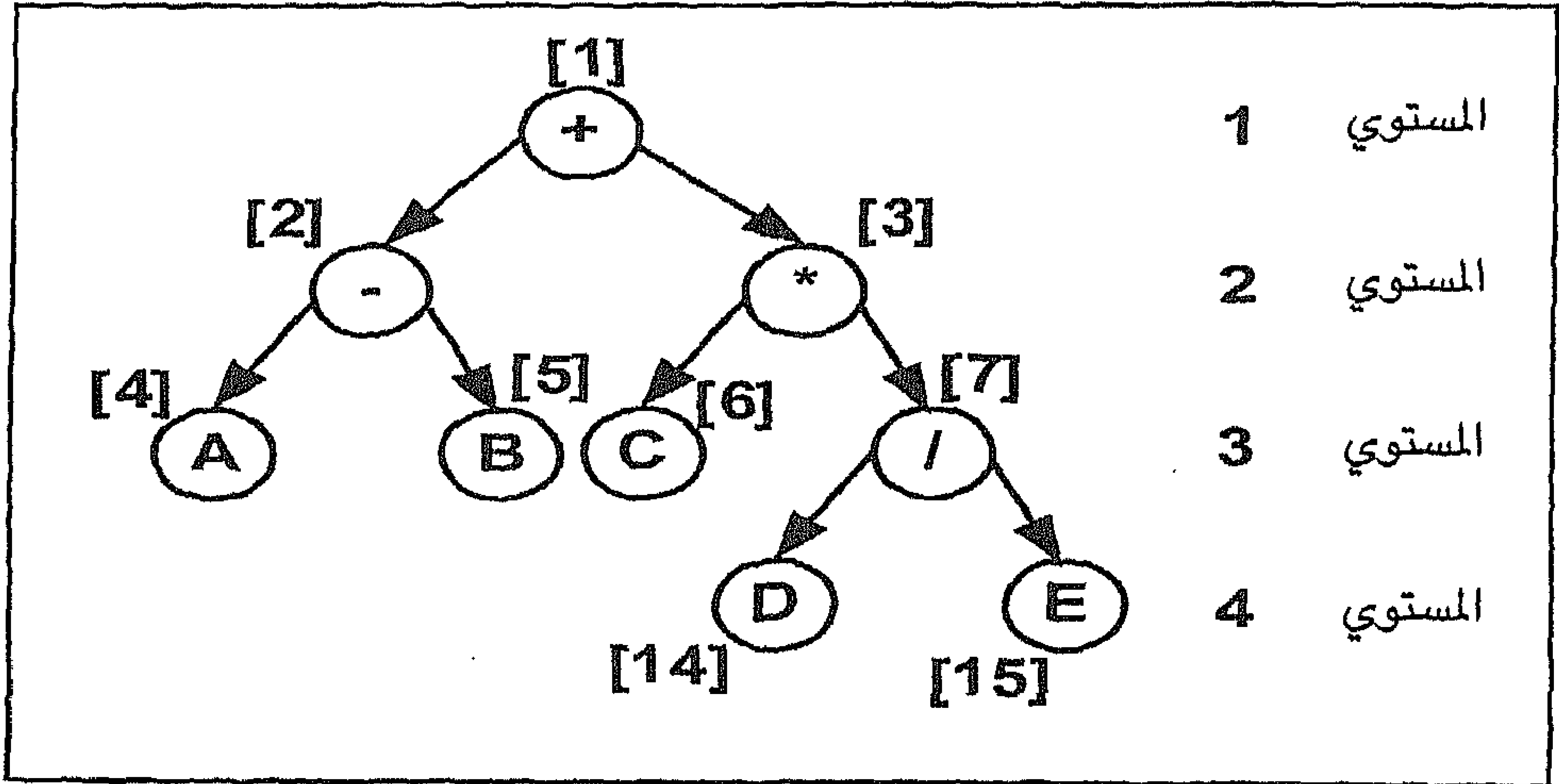
توجد طريقتان رئيستان لتمثيل الهياكل الشجرية الثنائية داخل ذاكرة الحاسوب هما: المصفوفات ذات البعد الواحد (One Dimensional Arrays) والقوائم المتصلة (Linked List Representation).

أولا: التمثيل باستخدام المصفوفات ذات البعد الواحد

يطلق على هذا النوع من تمثيل الهياكل الشجرية الثنائية التمثيل الخطي (Linear Representation) حيث تستخدم مصفوفة أو أكثر من المصفوفات ذات البعد الواحد. ويمكننا بيان ثلاث طرق مختلفة في هذا النوع من التمثيل، وذلك كما يلي:

تستعمل في الطريقة الأولى مصفوفة ذات بعد واحد تحتوي على عدد من المواقع مساويا (2^d-1)، حيث أن d تمثل عمق الهيكل الشجري الثنائي. ويأتي هذا العدد نتيجة لاحتواء كل عنصر من عناصر الهيكل الشجري الثنائي على ولدين على الأكثر، حيث أنه يمكن التأكد بأن أكبر عدد من العناصر الذي يحتوي عليه هيكل شجري ثنائي عمقه d هو (2^d-1) . ويمكنك إثبات ذلك والتأكد منه بسهولة.

على سبيل المثال افرض أن لدينا الهيكل الشجري الثنائي التالي (الشكل (8)):



الشكل (8)

والذي يمثل التعبير الحسابي:

$$(A - B) + C * (D / E)$$

ونلاحظ أن عمق هذا الهيكل الشجري = 4 . وبناء على ذلك نحتاج إلى مصفوفة مكونة من (2^4-1) أو 15 عنصراً لتمثيل جميع عناصر الهيكل الشجري.

ونلاحظ من الشكل (8) أيضاً أن الجذر يحتوي على عملية الجمع (+)، وأن عمليتي الطرح والضرب تقعان في المستوى الثاني، وأن العناصر الممثلة بالرموز / ,A,B,C تقع على المستوى الثالث، وأخيراً تقع العناصر التي تمثل بالرموز D,E في المستوى الرابع.

ونلاحظ أيضاً أن العناصر A, B, C ليس لها أبناء مع أنها تقع في مستوى قبل المستوى الأخير.

وفي طريقة التمثيل هذه يتم إعطاء أرقام متسلسلة لعناصر الهيكل الشجري الثنائي بدءاً بالجذر ثم بالعناصر التي تقع في المستوى الثاني ثم العناصر التي تقع في المستوى الثالث وهكذا، بحيث يتم الترقيم للعناصر الواقعة في المستوى نفسه، من اليسار إلى اليمين. ويعطينا هذا الترقيم ما يسمى بالهياكل الشجيرية الثنائية الكاملة، والتي سنتطرق إليها في الأجزاء القادمة من هذه الوحدة. ففي هذا النوع من الهياكل الشجرية ينبثق من كل عنصر من العناصر الواقعة في المستويات الثاني وإلى المستوى قبل الأخير عنصران اثنان.

ولو فرضنا أن اسم المصفوفة هو T فإنه يتم تخزين عناصر الهيكل الشجري في المصفوفة على النحو التالي:

1. يخزن الجذر في أول موقع من مواقع المصفوفة، ويشار إليه في لغة سي++ بالموقع T[1].
2. بالنسبة للعنصر المخزن في الموقع T[2]، فإن الابن الأيسر (إن وجد) يجب تخزينه في الموقع T[2 * i]
3. وبالنسبة للعنصر المخزن في الموقع T[i]، فإن الابن الأيمن (إن وجد) يجب أن يخزن فـي الموقع T[2 * i + 1]

وبالرجوع إلى الشكل (8)، وتطبيقاً لهذه القواعد نحصل على الأرقام المتسلسلة للعناصر كما هو واضح إلى يسار العناصر. ونرى كذلك أن العنصر الذي يحتوي على عملية القسمة (/) يحمل الرقم (7)، بينما العنصر D يحمل الرقم (14)، ذلك أن هذا العنصر يمثل الابن الأيسر للعنصر الذي يحتوي على عملية القسمة.

وبناءً على ما تقدم فإنه يتم تمثيل عناصر الهيكل الشجري الثنائي الوارد في الشكل (8) في المصفوفة T كما هو مبين في الشكل (9).

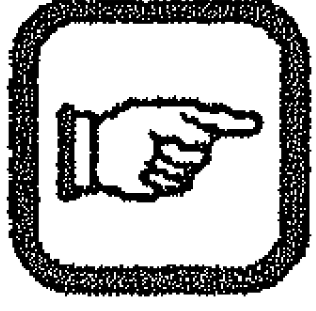
+	-	*	A	B	C	/							D	E
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

الشكل (9)

ومن الشكل (9) يمكننا الربط بين أي عنصر وأبناؤه وكذلك بين أي عنصر ووالده وذلك باستخدام العلاقات التالية:

- إذا كان موقع عنصر معين هو i فإن موقع ابنه الأيسر (إن كان موجوداً) هو (2*i).
- إذا كان موقع عنصر معين هو i فإن موقع ابنه الأيمن (إن وجد) هو (2*i+1).
- إذا كان موقع عنصر معين هوi فإن موقع والده هو INT(i/2) حيث أن INT تمثل الناتج الصحيح من خارج قسمة i/2 مع إهمال الكسر.

فعلى سبيل المثال، وبالرجوع إلى الشكل (9)، يمكننا معرفة تركيب الهيكل الشجري وإعادة بناءه ثانية. فالموقع الأول من المصفوفة يحتوي على الجذر، ولذلك نستطيع معرفة أن عملية الجمع تمثل جذر الشجرة. وبما أن موقع الجذر هو الموقع (1) فيمكننا معرفة الابن الأيسر له والذي هو الموقع (1*2) والذي يعطينا الموقع (2)، أي أن عملية الطرح تمثل الابن الأيسر لعملية الجمع، وكذلك فإن الابن الأيمن لهذه العملية يمكن تحديد موقعه باستخدام العلاقة (1+1*2) وهذا يعطينا الموقع (3) الذي يحتوي على عملية الضرب. وبالاستمرار بالطريقة نفسها يمكن معرفة تركيب الهيكل الشجري.



مثال (2)

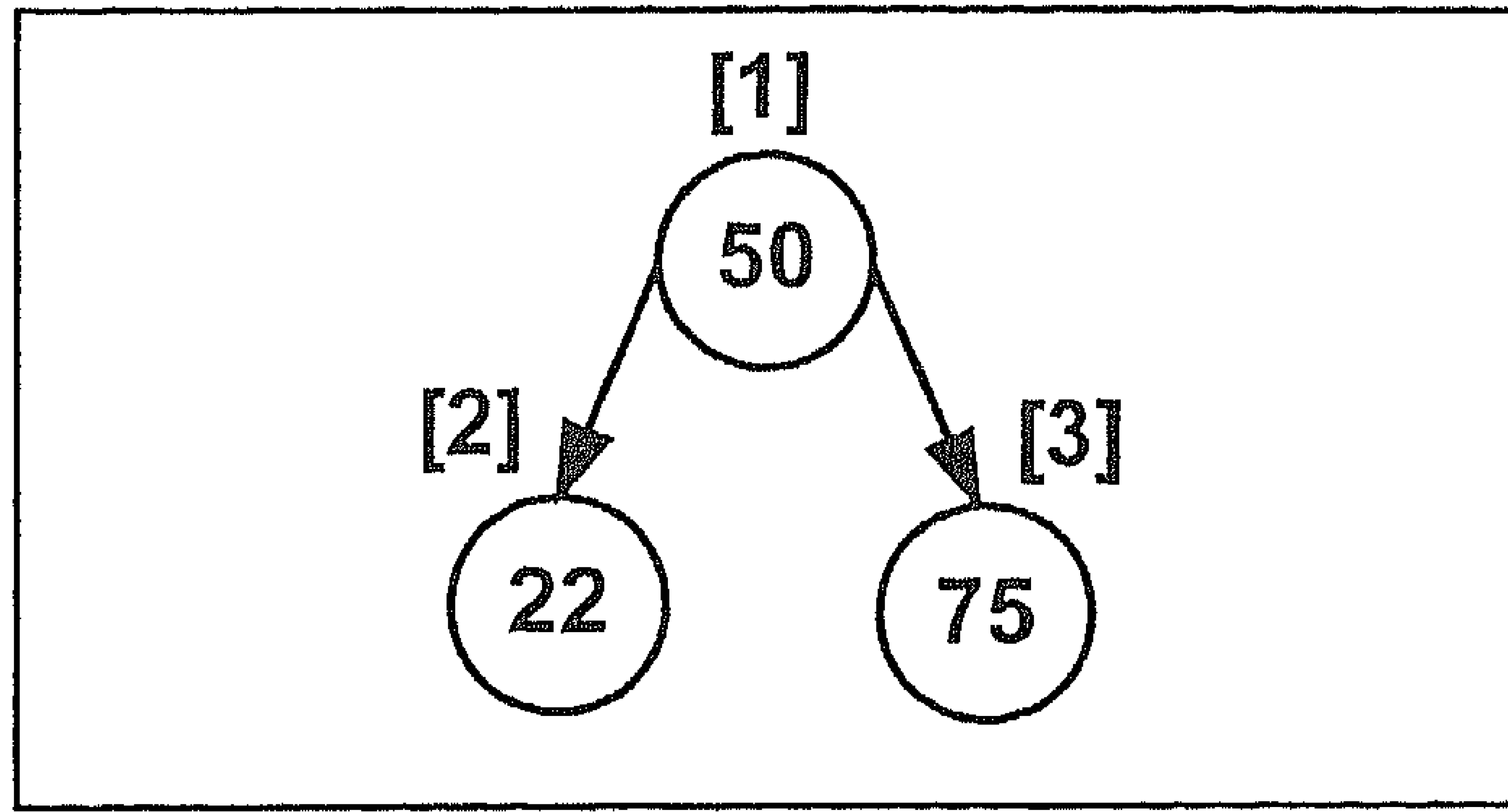
افرض أنك أعطيت التمثيل التالي الشكل (10) لهيكل شجري ثنائي، ارسم الهيكل الشجري:

50	22	75	11	60		90		15	25					45
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

الشكل (10)

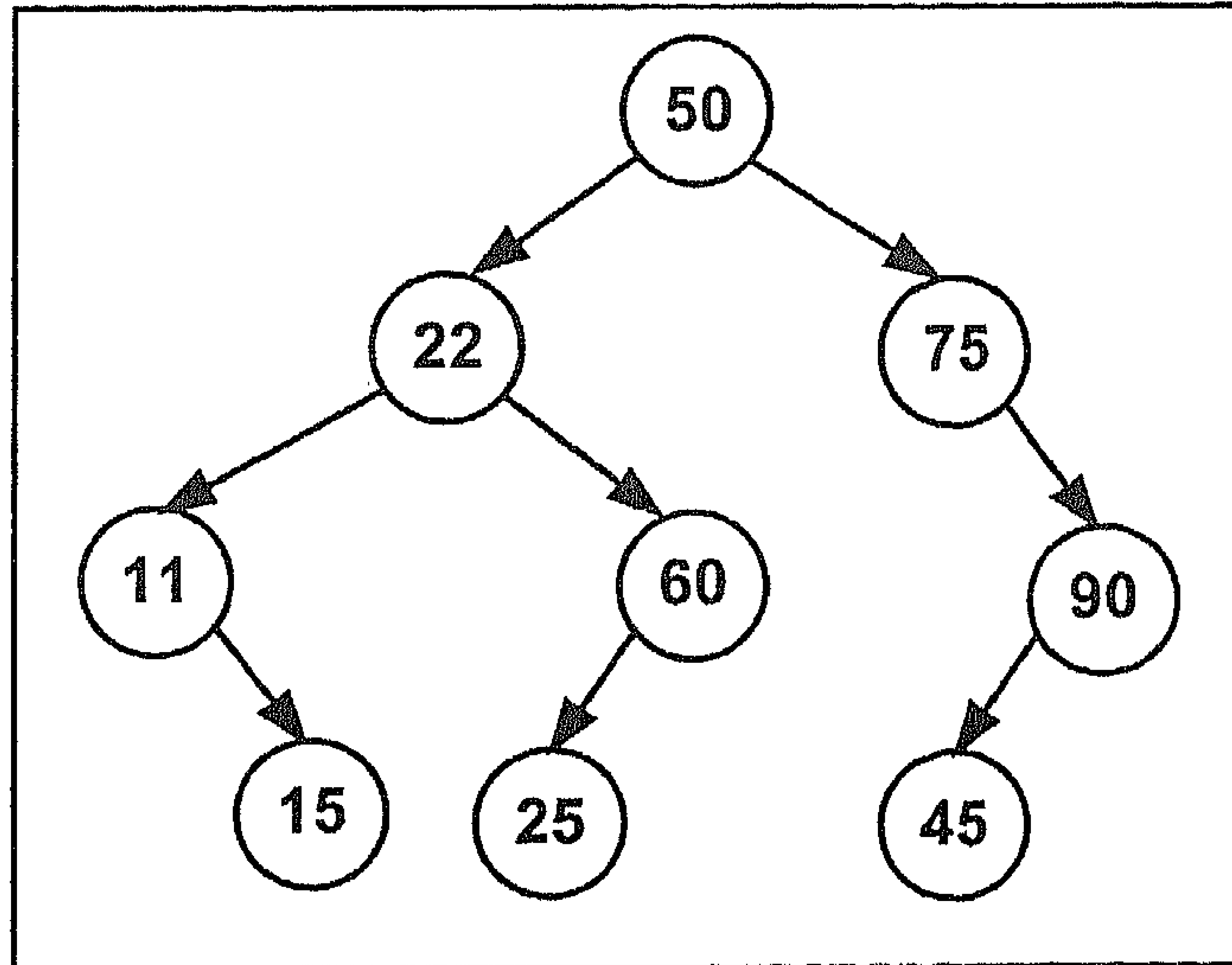
الحل:

بما أن الموقع الأول يحتوي على القيمة 50 فتمثل هذه القيمة جذر الشجرة، ويمكننا الآن معرفة الابن الأيسر للجذر وهو (1*2)، وهذا يعطينا القيمة 22. وكذلك فإن موقع الابن الأيمن للجذر هو (1 + 1*2) = 3، وهذا يعطينا القيمة 75 فنحصل على التركيب المبين في الشكل (11):

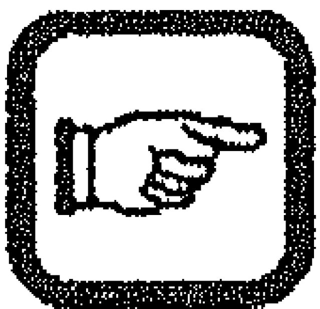


الشكل (11)

ونستمر هكذا بتحـديد موقع الابن الأيسر للعنصر الذي يحتوي على القيمة 22 وهو (2 * 2) أي الموقع الرابع حيث يحتوي هذا الموقع على القيمة 11. وكذلك نحدد موقع الابن الأيمن لهذا العنصر وهو (2 * 2 + 1) أي الموقع الخامس الذي يحتوي على القيمة 60 . وبالطريقة ذاتها نحصل على الهيكل الشجري التالي (الشكل (12)):

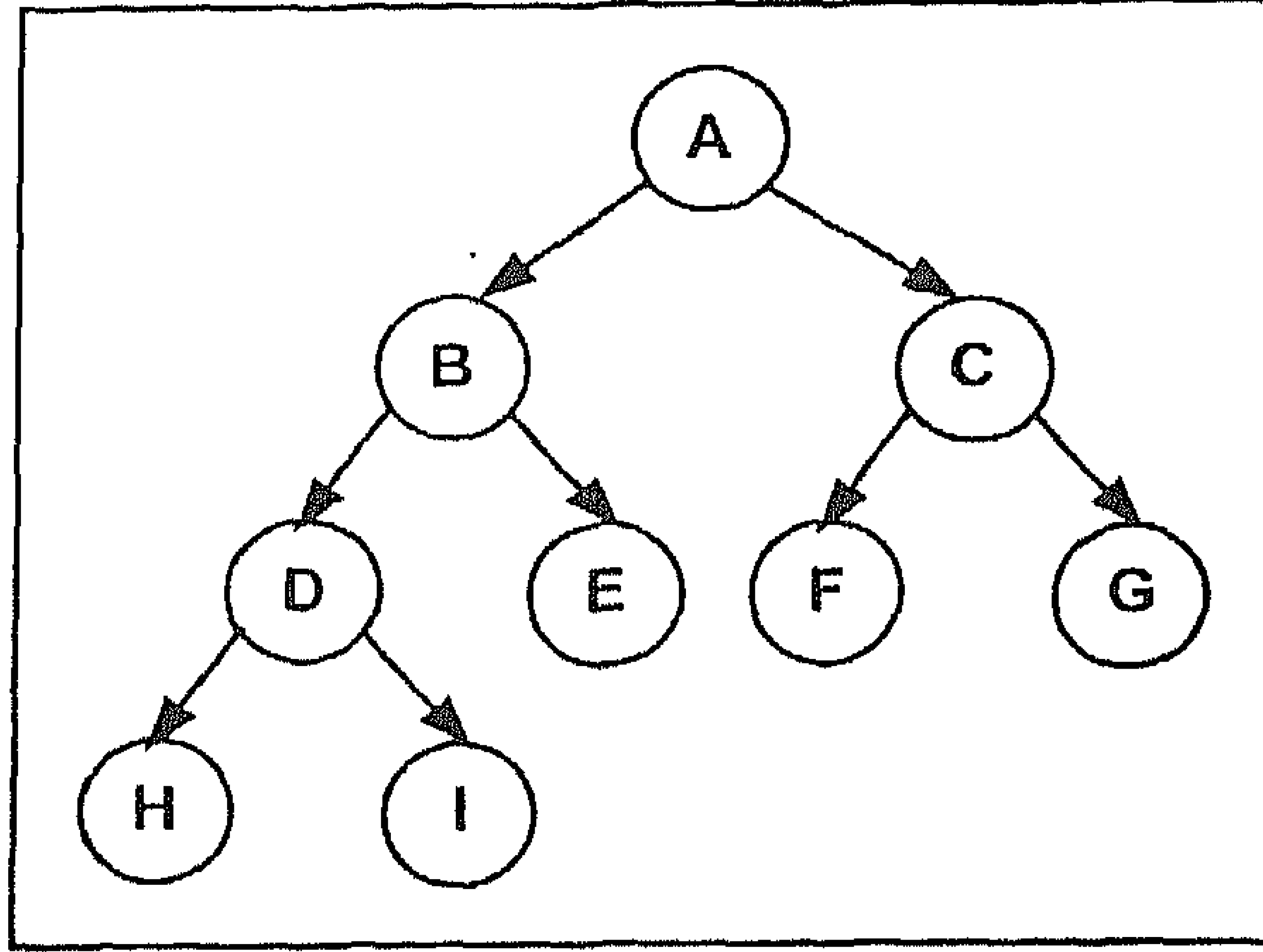


الشكل (12)



مثال (3)

وضح التمثيل الخطي للهيكل الشجري الثنائي التالي (الشكل (13)):



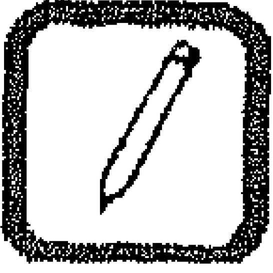
الشكل (13)

الحل:

كما هو مبين في الشكل (14):

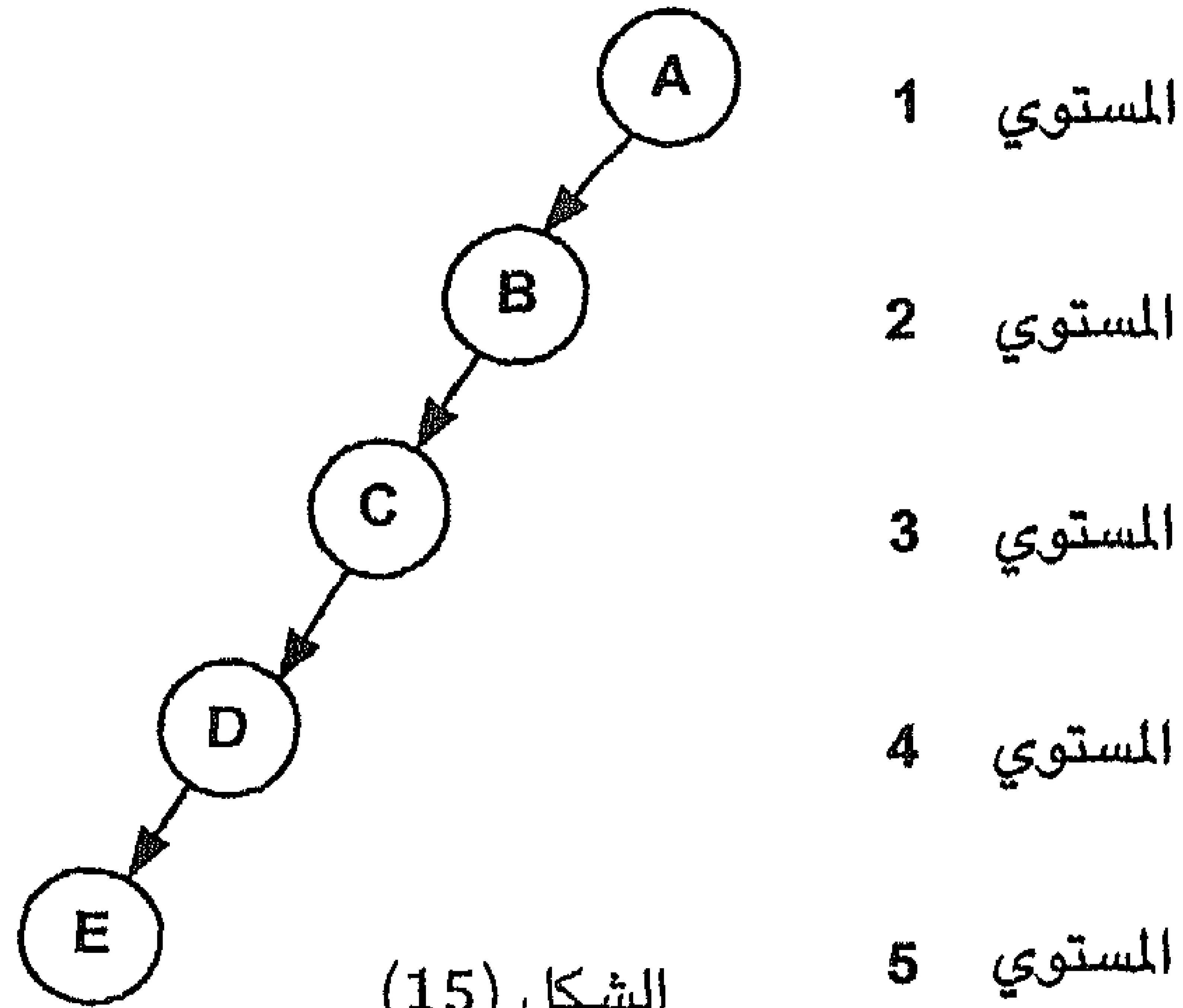
A	B	C	D	E	F	G	H	I	....	
1	2	3	4	5	6	7	8	9		15

شكل (14)



تدريب (2)

ما التمثيل الخطي للهيكل الشجري الثنائي التالي (الشكل 15):



الشكل (15)

أما الطريقة الثانية لتمثيل الهيكل الشجري الثنائي باستخدام مصفوفات فتعتمد على استعمال قيم عددية كمؤشرات لتمثيل الحواف (Edges)، ويمكن إنجاز ذلك في لغة باسكال كما هو مبين في التعريف التالي:

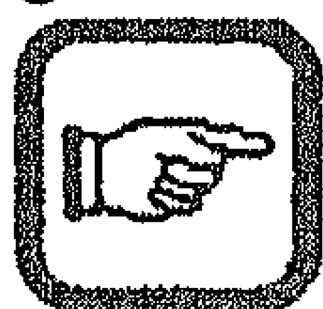
```
const    treeSize = 100;
typedef int  edge;
typedef  char elementType;

typedef struct
        {elementType info;
         edge  left;
         edge  right;
         }node ;

node    binaryTree[treeSize];
```

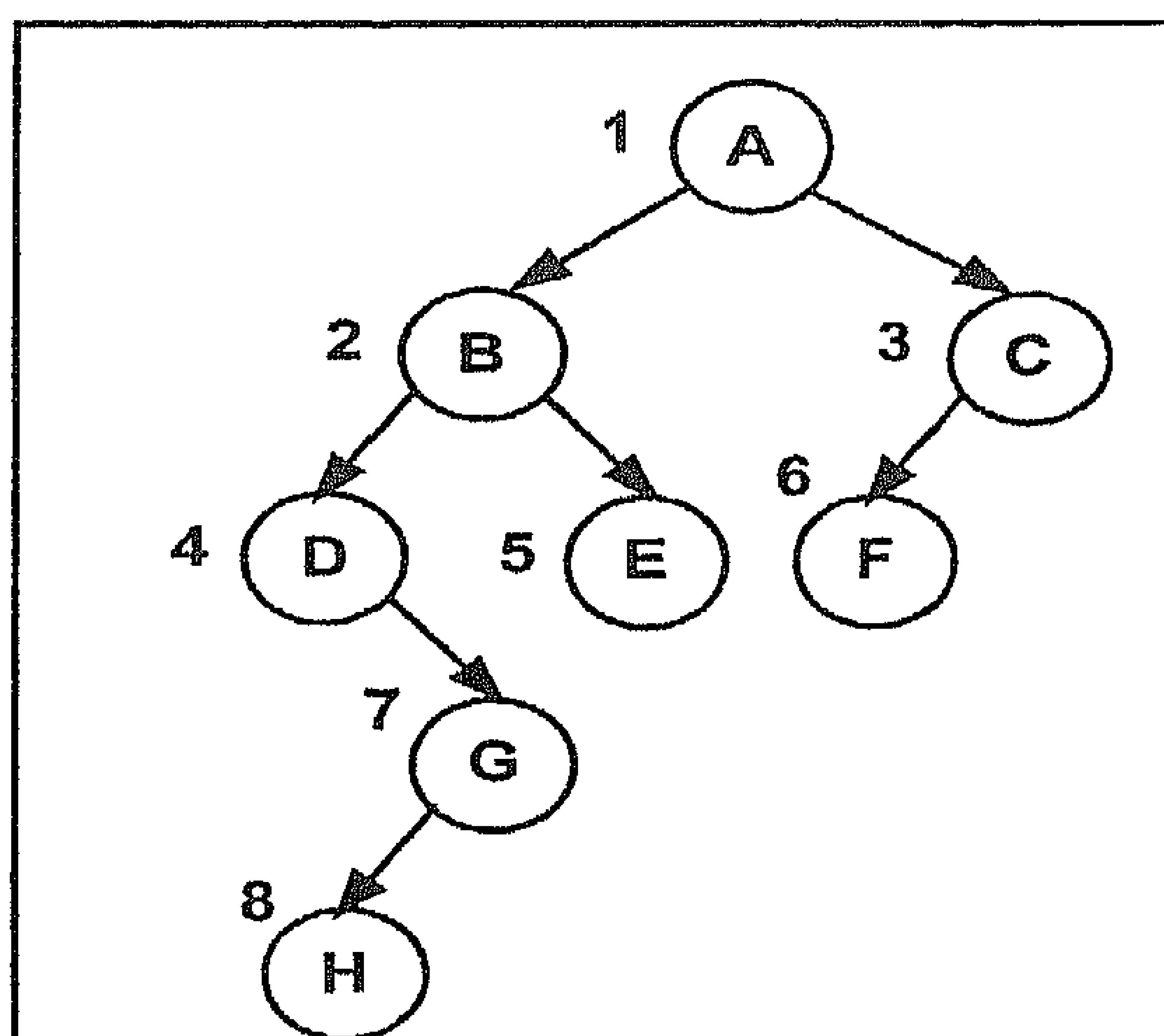
ففي هذا التعريف استخدمنا الثابت العددي treeSize ليحدد عدد عناصر الهيكل الشجري وهو يعتمد على المسألة المعطاة، وكذلك استخدمت القيمة 0 لتعني عدم وجود شجيرة فرعية (Empty subTree).

وفي تعريف العنصر استخدمنا (info) لتعني البيانات ذات الاهتمام وأنها من النوع (elementType).



مثال (4)

افرض أن لدينا الهيكل الشجري الثنائي التالي (الشكل 16):



الشكل (16)

فإنه يمكن تمثيله باستخدام مواقع المصفوفة لتمثيل الحواف كما يلي:

Index	Info	Left	Right
1	A	2	3
2	B	4	5
3	C	6	0
4	D	0	7
5	E	0	0
6	F	0	0
7	G	8	0
8	H	0	0

في هذه الحالة نحتاج إلى مصفوفة من السجلات يحتوي كل سجل على ثلاثة حقول ((Info و(Left) و(Right) أو إلى ثلاثة مصفوفات ذات بعد واحد، يحتوي كل منها على 8 مواقع. تمثل المصفوفة الأولى (Info) المعلومات التي نرغب عادة باسترجاعها من الهيكل الشجري. أما المصفوفتان (left) و(Right) فتحتويان على مؤشرات تشير إلى مواقع الابن الأيسر والابن الأيمن للعنصر الموجود في المصفوفة (Info).

فعلى سبيل المثال، وبالرجوع إلى المصفوفات الثلاث، نلاحظ أن Index تمثل مواقع عناصر المصفوفـــة (Info)، حيث أن (A) تقع في الموقع الأول من المصفوفة (Info)، وأن B تقع في الموقع الثاني والعنصر C يقع في الموقع الثالث وهكذا. وهذا موضح أيضاً في الهيكل الشجري المبين في الشكل (16). أما عملية ترقيم الهيكل الشجري فتتمثل في إعطاء الرقم (1) (أي الموقع الأول في المصفوفة Info) إلى العنصر (C) وهكذا، بحيث يتم إسناد الرقم (8) إلى العنصر (H) لوجود ثمانية عناصر فقط في الهيكل الشجري. وبالنسبة لعملية ترقيم الهيكل الشجري فتتم عن طريق أخذ المستوى الأول (أي الجذر) ومن ثم المستوى الثاني من اليسار إلى اليمين ومن ثم إلى المستوى الثالث، وهكذا.

أما عن كيفية تحديد محتويات المصفوفتين (Left) و(Right) أي مصفوفات الابن الأيسر والابن الأيمن للعناصر) فيتم ذلك بالرجوع إلى الهيكل الشجري. فتلاحظ من الهيكل الشجري أن (B) يمثل الابن الأيسر للعنصر (A) . وبما أن موقع العنصر (B) هو الثاني في المصفوفة (Info)، تجد أن محتويات الموقع الأول من المصفوفة Left (أي موقع الابن الأيسر للعنصر A) هي (2) (أي موقع B في المصفوفة Info).

ولتوضيح ذلك أكثر، لو أخذنا العنصر (D) في الهيكل الشجري نجد أنه أخذ الرقم 4 (أي أن الموقع الرابع في المصفوفة(left) يمثل مؤشر الابن الأيسر للعنصر (D) "وهو (O) في هذه الحالة لعدم وجود ابن أيسر للعنصر (D) كما هو موضح في الهيكل الشجري") وأن القيمة المخزنة في الموقع الرابع للمصفوفة Right يمثل مؤشر الابن الأيمن للعنصر D (وهو 7 في هذه الحالة والذي يشير إلى الموقع السابع في المصفوفة Info والذي يحتوي على القيمة G). أي أن G هو الابن الأيمن للعنصر D كما هو مبين أيضاً من الهيكل الشجري المبين في الشكل (16).

وبهذه الطريقة نستطيع تمثيل جميع عناصر الهيكل الشجري الثنائي باستخدام ثلاث مصفوفات كما بيّنا.

أما الطريقة الثالثة والأخيرة للتمثيل الخطي للهياكل الشجرية الثنائية باستخدام المصفوفات ذات البعد الواحد فتتمثل بتخزين العناصر في المصفوفة حسب إحدى طرق الاستعراض الثلاث المعروفة للهياكل الشجرية والتي سنتطرق لها في بحث العمليات على الهياكل الشجرية.

ففي هذه الطريقة نحتاج لتمثيل كل عنصر إلى خانتين ثنائيتين تستخدم الأولى لتدل إن كان لهذا العنصر ابن أيسر أم لا، وتستخدم الخانة الثنائية الأخرى للدلالة إن كان للعنصر ابن أيمن أم لا. ففي حالة وجود ابن أيسر تصبح قيمة الخانة الثنائية الدالة على ذلك True وإلا فإنها تصبح False، وهذا ينطبق أيضاً على الابن الأيمن.

ويمكن الإعلان عن هذا التمثيل بلغة سي++ كما يلي:

```
const    treeSize = 100;
typedef  char elementType;

typedef struct
        {elementType info;
         bool  left;
         bool  right;
         }node ;

node   binaryTree[treeSize];
```

فباستخدام هذه الطريقة يمكن تمثيل الهيكل الشجري في المثال (4) على النحو التالي باستخدام طريقة (PreOrder):

Index	Info	Left	Right
1	A	T	T
2	B	T	T
3	D	F	T
4	G	T	F
5	H	F	F
6	E	F	F
7	C	T	F
8	E	F	F

حيث T تعني TRUE وF تعني FALSE.

وتعتبر الطريقة الأولى من هذه الطرق الثلاث الأكثر انتشاراً واستخداماً من الطريقتين الأخريين.

وتجدر الإشارة، عزيزي الدارس، أن من سيئات تمثيل الهياكل الشجرية باستخدام المصفوفات هو عدم استغلال الذاكرة على الوجه الأمثل في كثير من الأحيان. يتضح ذلك من تمثيل الهياكل الشجرية الثنائية المنحرفة باتجاه واحد فقط كما هو الحال في تدريب (2). ففي هذه الحالة استغلت خمسة مواقع فقط من أصل ستة عشر موقعاً في المصفوفة.

والمشكلة الأخرى التي يعاني منها تمثيل الهياكل الشجرية بهذه الطريقة هو: عند إضافة العناصر أو حذفها إذ غالباً ما يتطلب ذلك إزاحة الكثير من العناصر للمحافظة على مستويات هذه العناصر وعلى العلاقة الجديدة بين العناصر مع بعضها البعض. وللتغلب على هذه المشكلات تستخدم الطريقة الثانية كما يلي:

ثانياً: تمثيل الهياكل الشجرية الثنائية باستخدام القوائم المتصلة (Linked List Representation)

ينقسم العنصر الواحد من عناصر الهيكل الشجري الثنائي في هذا التمثيل إلى ثلاثة أقسام هي:

1. قسم البيانات (data)
2. الابن الأيسر (LeftChild)
3. الابن الأيمن (RightChild)

أما القسمين الثاني والثالث فيُمثلان مؤشرات الربط بين العنصر مع الابن الأيسر والابن الأيمن المنبثقين من هذا العنصر.

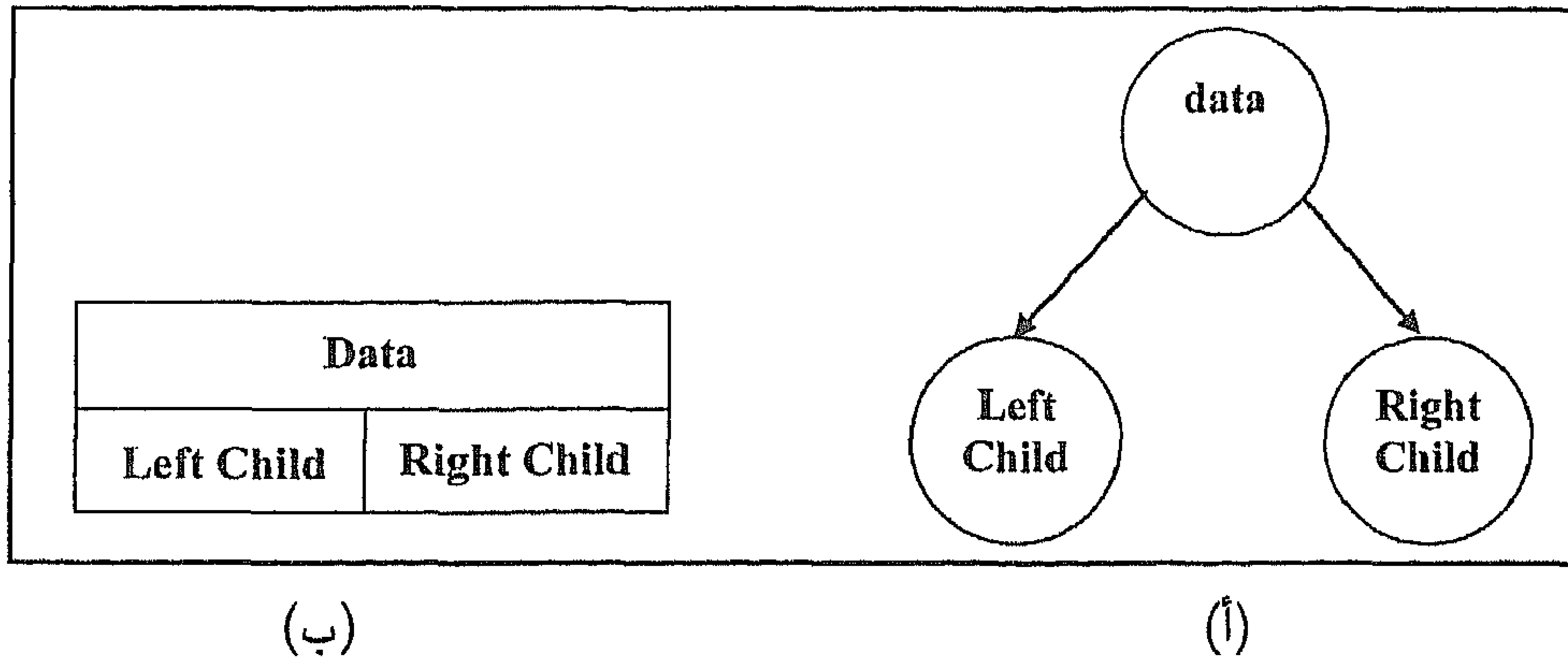
ويمكن تعريف العنصر من هذه العناصر في لغة باسكال كما يلي:

```
typedef  char elementType;

typedef struct node *nodePtr;
struct node
        {elementType data;
         nodePtr LeftChild,RightChild;
        };
```

حيث يمكن أن يكون نوع البيانات (data) أياً من الأنواع البسيطة أو المركبة مثل الأنواع الصحيحة أو الحقيقية أو الرمزية أو سجلاً بحد ذاته وما إلى ذلك.

ويمكن بيان مثل هذا العنصر المعرف أعلاه بإحدى طريقتي التمثيل التاليتين (الشكل (17 - أ، ب)):



(ب) (أ)

الشكل (17)

على الرغم أنه من الصعب في هذا التمثيل تحديد الوالد أو الأخوة لعنصر معين والذي يعتبر سهلاً في حالة استخدام مصفوفة ذات بعد واحد، إلا أنه يعتبر كافياً لمعظم التطبيقات التي سنتطرق إليها. وفي حالة الضرورة لتحديد الوالد أو الأخوة لعنصر معين، في الهيكل الشجري، فإنه يمكن إضافة حقول إضافية أخرى للعنصر تحدد الوالد والأخ التالي. ولا بد من الإشارة إلى أنه يشار إلى الهيكل الشجري باسم المتغير الذي يشير إلى الجذر، حيث أنه بدأ بالجذر، ويمكن الوصول إلى جميع عناصر الهيكل الشجري.

4. العمليات الأساسية على الهياكل الشجرية الثنائية

يمكن إجراء العمليات التالية على الهياكل الشجرية الثنائية:

1. إنشاء الهيكل الشجري الثنائي (Binary Tree Creation).
2. استعراض الهيكل الشجري الثنائي (Binary Tree Traversal).
3. البحث عن عنصر معين في الهيكل الشجري الثنائي (Binary Tree Search).
4. الإضافة إلى الهيكل الشجري الثنائي (Binary Tree Insertion).
5. الحذف من الهيكل الشجري الثنائي (Binary Tree Deletion).
6. ترتيب القيم المخزنة في الهيكل الشجري الثنائي (Binary Tree Sort).

بعد التعرف على طرق تمثيل الهيكل الشجري الثنائي داخل ذاكرة الحاسوب للتمكن من إجراء العمليات المناسبة عليها، سنتطرق إلى إجراء هذه العمليات مستخدمين مؤشرات الربط (Pointers) لكون هذا الأسلوب من أساليب التمثيل يتميز بالديناميكية والمرونة وسهولة إجراء العمليات، حيث أنه يمكن الإضافة إلى الهيكل الشجري الثنائي والحذف منه بسهولة كبيرة. وسنستخدم التعريف التالي في توضيح العمليات الآنفة الذكر على الهياكل الشجرية الثنائية في البرامج والدوال اللازمة لبيان العمليات المختلفة والممكنة على الهياكل الشجرية:

```
class treePtr
{private:
        treePtr * leftChild;
        treePtr * rightChild;
        int info;
 char ch;
public:
   treePtr*  treeInit(treePtr*);
  treePtr*  insertNode(treePtr*,int val);
  ..........
};
```

وكما هو واضح من هذا التعريف، عزيزي الدارس، فإن نوع البيانات المراد تخزينها في عناصر الهيكل الشجري (أي نوع Info) هو عددي صحيح (int) . ومن الممكن أن تكون هذه البيانات من الأنواع البسيطة أو المركبة، وللتسهيل على فهم هذه العمليات استخدمنا نوعاً بسيطاً من البيانات وهو النوع العددي الصحيح.

1.4 إنشاء الهيكل الشجري الثنائي (Binary Tree Creation)

سوف نستعرض في هذا الجزء موضوع إنشاء هيكل شجري تحتوي عناصره على أعداد صحيحة، على فرض أن هذه الأعداد مخزنة في ملف يحتوي كل سطر من سطوره على عدد صحيح ما. وسوف يتم تخزين كل عدد من الأعداد في عنصر منفصل من عناصر الهيكل الشجري، وكذلك فإنه سيتم تخزين الأعداد بحيث يمكننا طباعتها مرتبة عند تطبيق إجراء استعراض الهيكل الشجري البيني (Inorder Traversal).

ولإنجاز ذلك، تتطلب عملية التخزين أن يتم تخزين القيمة الأكبر في الجهة اليمنى للوالد وتخزين القيمة الأصغر في الجهة اليسرى للوالد. بحيث يمتلك الهيكل الشجري في محصلة الأمر الخاصية التالية: كل عنصر في الشجيرة الفرعية اليمنى يكون أكبر من قيمة أي عنصر في الشجيرة الفرعية اليسرى والجذر نفسه يكون قيمته أكبر من أي عنصر في الشجيرة اليسرى وأصغر من أي عقدة في الشجيرة اليمنى وهذه الخاصية تنطبق على أي شجيرة فرعية.

يمكن تقسيم برنامج إنشاء الهيكل الشجري الثنائي إلى الأجزاء التالية:

أ- الجزء التمهيدي:

ويتكون من دالة سنطلق عليها الاسم (treeInit)، حيث يتم في هذه الدالة إنشاء عنصر الجذر، ثم قراءة العدد الأول من قائمة أعداد ندخلها من لوحة المفاتيح، وتخزين هذا العدد في الجذر. ويتم كذلك في هذه الدالة إسناد القيمة NULL)) إلى كل من مؤشر الابن الأيسر (Left child pointer) ومؤشر الابن الأيمن (Right child pointer) في عنصر الجذر.

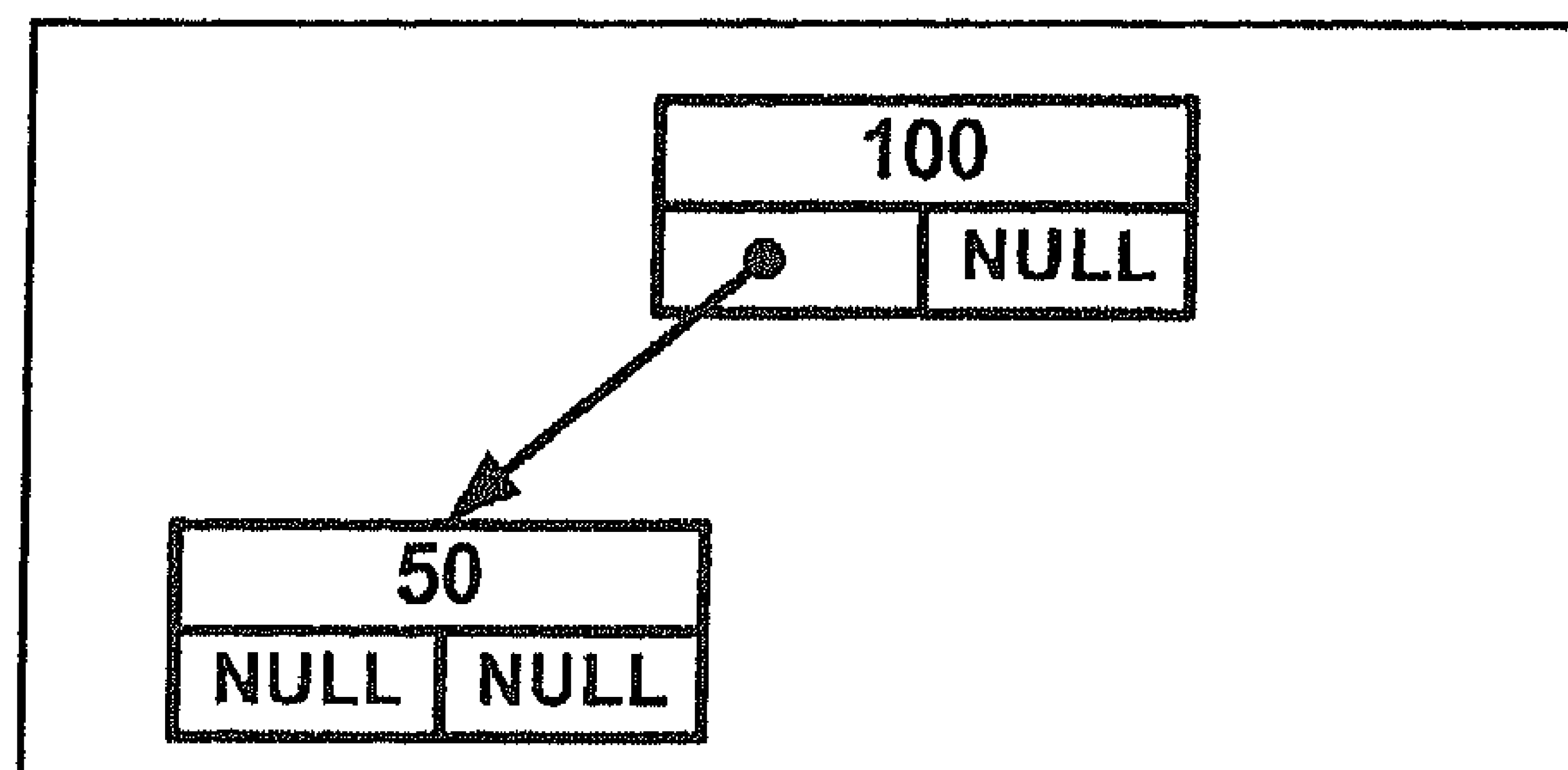
ب- الدالة الثانية المستخدمة والتي تحمل الإسم insertNode:

سوف تتولى مسؤولية إدخال القيم إلى عناصر جديدة في الهيكل الشجري وفي المواقع الصحيحة لها حسب الخاصية السالف ذكرها.

ويمكن تلخيص عمل هذه الدوال كما يلي:

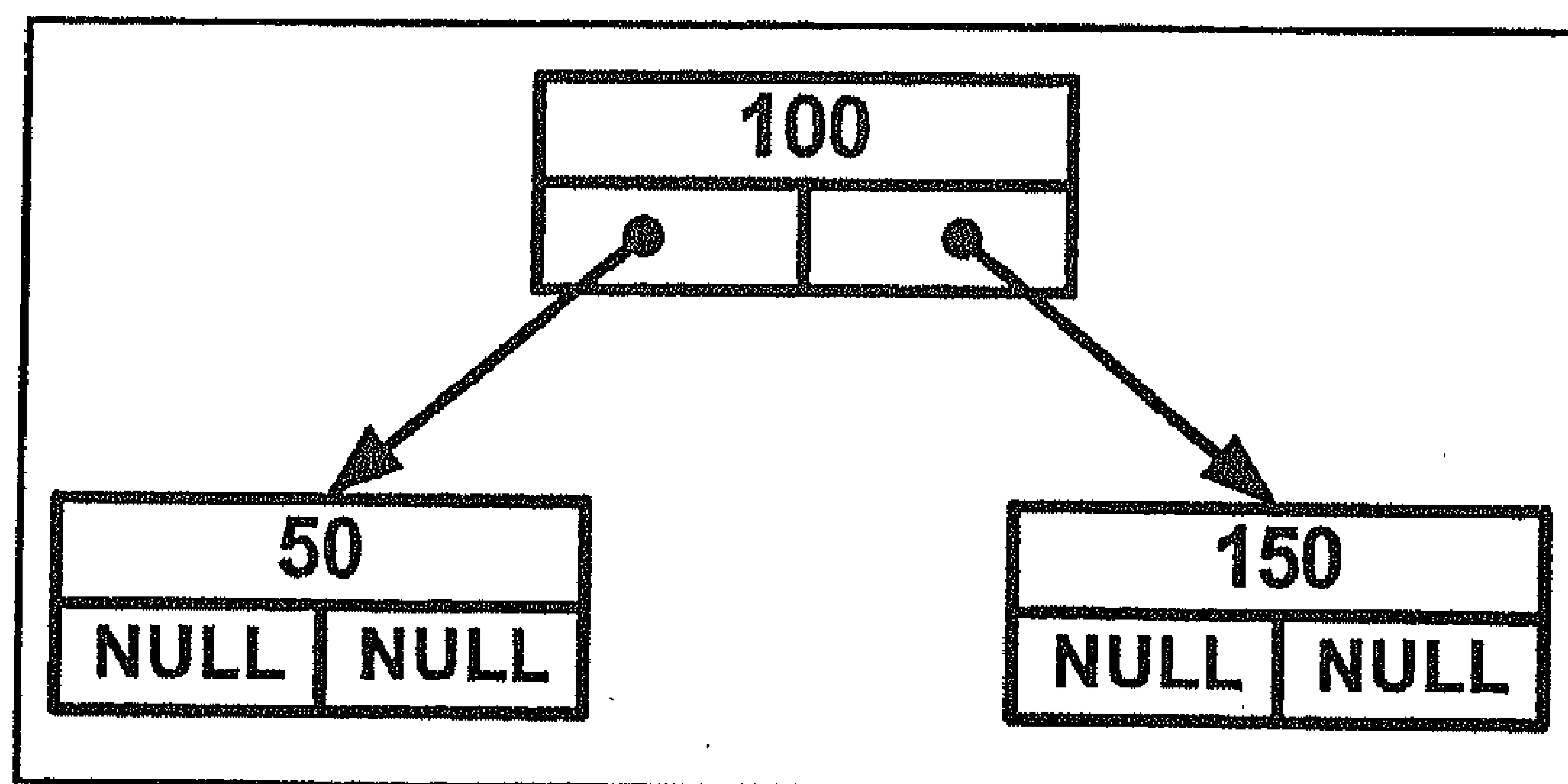
بعد أن نكون قد خزنّا القيمة الأولى في الجذر عن طريق الإجراء (treeInit) نقوم بقراءة القيمة التالية من لوحة المفاتيح عن طريق جملة قراءة عادية. ويمكن أن تتم عملية القراءة من داخل البرنامج الرئيسي، وليس بالضرورة من داخل الدالة ((insertNode، مع العلم أنه يمكن إتمام هذه العملية من داخل الدالة.

افرض على سبيل المثال أن القيمة التي خزنت في الجذر تساوي "100" وأن القيمة الثانية هي " 50"، فيتم مقارنة القيمة "50" مع القيمة "100"، فإذا كانت هذه القيمة (أي القيمة 50 والتي تعتبر الآن بمثابة القيمة التالية) أصغر من القيمة المخزنة في الجذر، فننتقل إلى الشجيرة اليسرى (Left Subtree) للجذر، وبما أن هذه الشجيرة لا تحتوي على عناصر فإن مؤشر الابن الأيسر للجذر سيؤشر على هذا العنصر الجديد، بحيث نحصل على الشكل التالي بعد أن تخزن القيمة (NULL) في مؤشر الابن الأيمن ومؤشر الابن الأيسر للعنصر الذي يحتوي على القيمة "50". (الشكل (18))



الشكل (18)

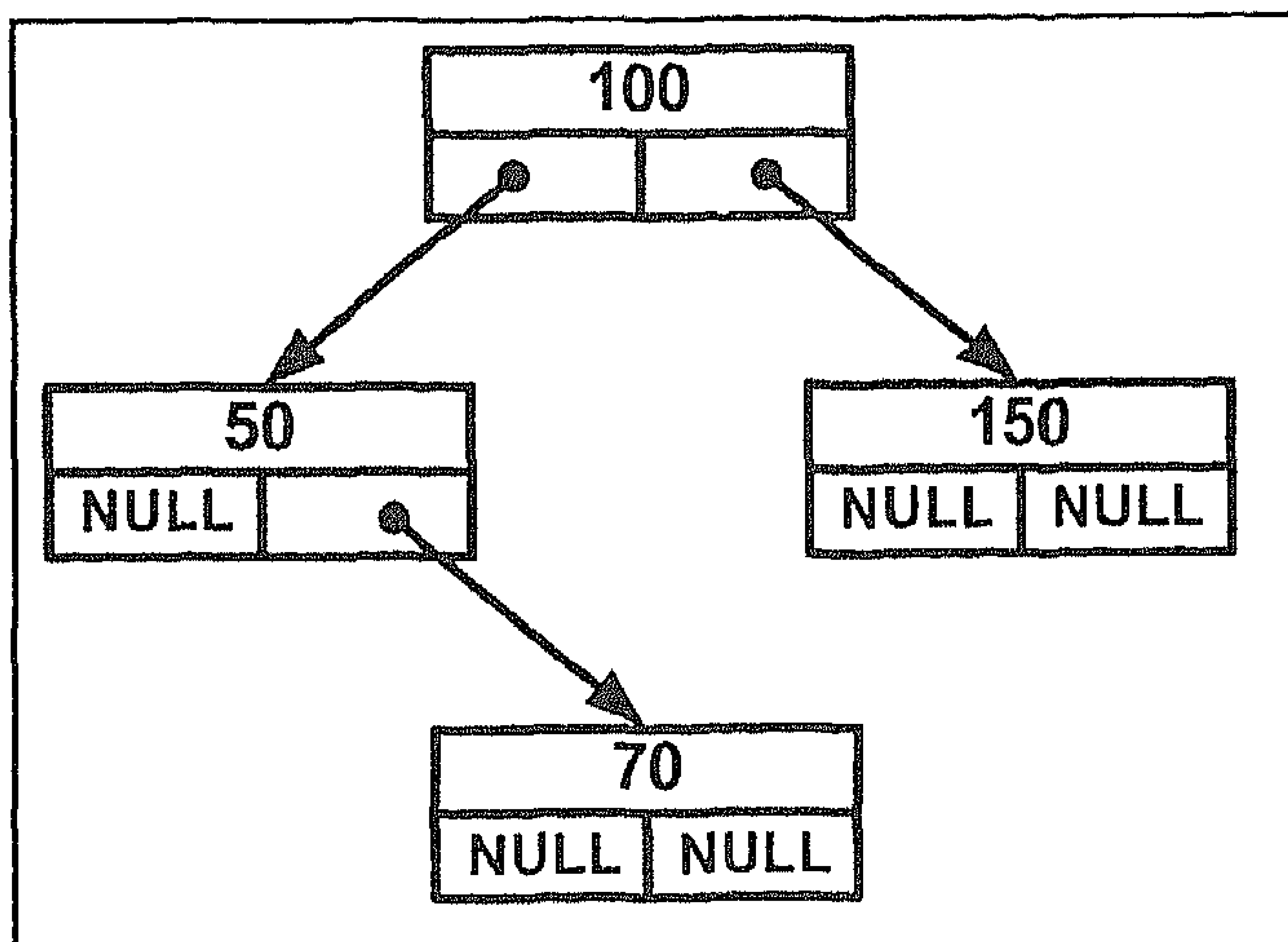
وأما إذا كانت القيمة التالية أكبر من أو مساوية للقيمة الموجودة في الجذر، فيتم المسير في اتجاه الشجيرة اليمنى، فعلى سبيل المثال لو فرضنا أن القيمة التالية (أي القيمة الثالثة) هي "150" فسوف يتم إضافة هذه القيمة في الشجيرة اليمنى (Right Subtree) للجذر، ونتيجة لذلك نحصل على الشكل (19) التالي:



الشكل (19)

وأخيراً، لو فرضنا أننا نود أن نضيف إلى هذا الشكل القيمة الأخيرة وهي "70"، فإننا نبدأ بمقارنة هذه القيمة مع الجذر، فبما أن القيمة "70" أصغر من القيمة "100" فإننا نعرف أن هذه القيمة يجب أن تقع في الشجيرة اليسرى للجذر.

بعد ذلك نقارن القيمة "70" مع القيمة "50". ولكون القيمة "70" أكبر من القيمة "50" فهذا يعني أن هذه القيمة يجب أن تقع في الشجيرة اليمنى بالنسبة للقيمة "50"، وبما أنه لا توجد عناصر في الشجيرة اليمنى للعنصر الذي يحوي القيمة "50" فإننا نحصل على الهيكل الشجري التالي (الشكل (20):



الشكل (20)

ونستمر بهذا الأسلوب لقراءة وإضافة جميع القيم إلى الهيكل الشجري، ويمكن تلخيص هذه الطريقة لإضافة القيم إلى الهيكل الشجري بمقارنة القيمة التالية مع القيمة المخزنة في جذر الشجرة، فإذا كانت هذه القيمة الجديدة أصغر فهذا يعني بأن القيمة الجديدة يجب أن تقع في الشجيرة اليسرى (Left Subtree) للجذر. ولهذا فإننا نفحص بعد ذلك قيمة المؤشر الأيسر للجذر، فإذا كانت قيمته NULL، فيتم تعديل هذه القيمة لكي يشير إلى العنصر الجديد الذي يحوي القيمة الجديدة، أما إن لم تكن قيمة المؤشر الأيسر للجذر مساوية للقيمة (NULL) فإنه يتم الانتقال إلى العنصر المؤشر عليه بالمؤشر الأيسر للجذر، وبعد ذلك نقارن القيمة المقروءة مع القيمة الموجودة مع هذا العنصر. وهذا يعيدنا إلى نقطة البداية حيث يعامل هذا العنصر وكأنه جذر الهيكل الشجري حيث ينطبق عليه ما تم تفصيله عند مقارنة القيمة مع جذر الهيكل الشجري.

أما إذا كانت القيمة الجديدة التي تم قراءتها أكبر من أو مساوية للقيمة في أي من عناصر الهيكل الشجري الذي نقارن القيمة الجديدة معه، فإننا بدل أن نسير باتجاه الشجيرة اليسرى لهذا العنصر فإننا نذهب إلى الشجيرة اليمنى.

ولهذا فإن عملية إضافة قيمة معينة تتلخص في مقارنة هذه القيمة مع القيم الموجودة في العناصر الموجودة في الهيكل الشجري، وبناء على نتيجة المقارنة فإما أن تتبع الشجيرة اليسرى أو الشجيرة اليمنى للعنصر الذي تتم المقارنة معه، وتستمر هذه العملية لحين

مصادفة القيمة (NULL) لإحدى المؤشرات. ويعتبر العنصر الذي يحتوي على هذا المؤشر بمثابة الوالد للعنصر الذي يحتوي على القيمة الجديدة والذي سيضاف إلى الهيكل الشجري، بعدئذٍ يتم تعديل قيمة المؤشر من القيمة (NULL) لكي يشير على العنصر الجديد. ويلاحظ أن فكرة إيجاد الموقع الصحيح للعنصر التالي عند إضافته إلى الهيكل الشجري الثنائي المنشأ لهذه الطريقة تتلخص بتخزين القيمة الأكبر في الجهة اليمنى للوالد والقيمة الأصغر في الجهة اليسرى للوالد.

ج- أما الجزء الثالث لبرنامج إنشاء الهيكل الشجري الثنائي:

فيتمثل في نموذج التحكم في البرنامج، ويمثل هذا الجزء، الجزء الأخير في البرنامج والذي يتحكم باستدعاء الدالة (treeInit) والدالة (insertNode) . وفيما يلي،عزيزي الدارس، برنامج إنشاء الهيكل الشجري الثنائي:

```
#include <iostream.h>
#include <process.h>
class treePtr
  {treePtr * leftChild;
   treePtr * rightChild;
   int info;
  public:
     treePtr*  treeInit(treePtr*);
     treePtr*  insertNode(treePtr*,int val);
 };
/////////////////////////////////////////////////////
treePtr*  treePtr::treeInit(treePtr* treeRoot)
   {//treePtr *root;
    treeRoot=new treePtr;
    cin>>ch;
   treeRoot->info=ch;
   treeRoot->leftChild=NULL;
   treeRoot->rightChild=NULL;
   return treeRoot;
   }
//////////////////////////////////////////////////////
treePtr* treePtr::insertNode (treePtr* treeRoot,int val)
  { treePtr      *currentnode, *newnode;
```

```
bool    inserted=false;
newnode=new treePtr;
currentnode=treeRoot;
while (!inserted)
 {if (val < currentnode->info)
   if (currentnode->leftChild !=NULL)
    currentnode=currentnode->leftChild;
    else
     {currentnode->leftChild= newnode;
      inserted=true;
     }
   else
    if (currentnode->rightChild !=NULL)
     currentnode=currentnode->rightChild;
     else
      {currentnode->rightChild= newnode;
       inserted=true;
      }
 }
 newnode->info= val;
 newnode->leftChild=NULL;
 newnode->rightChild=NULL;
return treeRoot;
}
////////////////////////////////////////////////////////////
void main()
 {
        treePtr *root=NULL,d1;
         root=d1.treeInit(root);
        int choice,elm;
        while(1)
          {cout<<"\n********** treePtr LINK LIST
**********\n";
            cout<<"Choices Are:\n=>[1] For Insert \n=>[2] For Exit";
            cout<<"\nEnter Your choice: ";
            cin>>choice;
    switch (choice)
                 {case 1: { cin>>elm;
                             root=d1.insertNode(root,elm);
                           }   break;
        case 2: exit(0);
            }
        }
 }
```

وتجدر الإشارة إلى أن الهيكل الشجري الذي ينشأ من تنفيذ البرنامج السابق يتميز بخاصية أن القيمة الموجودة في أي من العناصر أكبر من جميع القيم الموجودة في العناصر التي تشكل الشجيرة اليسرى لذلك العنصر، وكذلك فإن هذه القيمة أصغر من أو مساوية للقيم الموجودة في العناصر التي تشكل الشجيرة اليمنى لهذا العنصر. ويطلق على هذا النوع من الهياكل الشجرية بشجيرة البحث الثنائي (Binary search Tree). ومن خصائص هذا النوع من الهياكل الشجرية أنه يمكن البحث (الاستقصاء) عن عنصر معين بسرعة كبيرة حيث يمكن تطبيق مبدأ البحث الثنائي (Binary Search).

2.4 استعراض الهيكل الشجري الثنائي (Binary Tree Traversal)

تعني هذه العملية زيارة كل عنصر من عناصر الهيكل الشجري مرة واحدة وذلك حسب ترتيب معين لهذه العناصر، ومن ثم يتم إجراء عملية محددة على بيانات هذه العناصر مثل زيادة راتب كل موظف أو إجراء تعديل معين على جميع علامات الطلبة، أو إيجاد المجموع الكلي لرواتب شركة معينة وما إلى ذلك. ويمكن استعمال إجراءات استعراض بسيطة وذلك باستخدام مبدأ الاستدعاء الذاتي (Recursion) والذي يعكس الطابع التكراري لبناء الهياكل الشجرية الثنائية وتعريفها، إذ إن كل شجيرة فيها يمكن النظر إليها على أنها هيكل شجري ثنائي بحد ذاتها.

ولتوضيح طرق الاستعراض المختلفة، افرض أن (L) تعني التحرك باتجاه مؤشر الابن الأيسر، وأن R تعني التحرك باتجاه مؤشر الابن الأيمن، وأن (D) تعني طباعة البيانات المراد بيانها والموجودة في العناصر المختلفة للهيكل الشجري، يمكن استخلاص ثلاث طرق متكررة (Recursive) لاستعراض عناصر الهيكل الشجري وهي:

LDR	(Left Data Right)
LRD	(Left Right Data)
DLR	(Data Left Right)

حيث يطلق على هذه الطرق الثلاث الأسماء المبينة إزاء كل منها والمعرفة كالتالي:

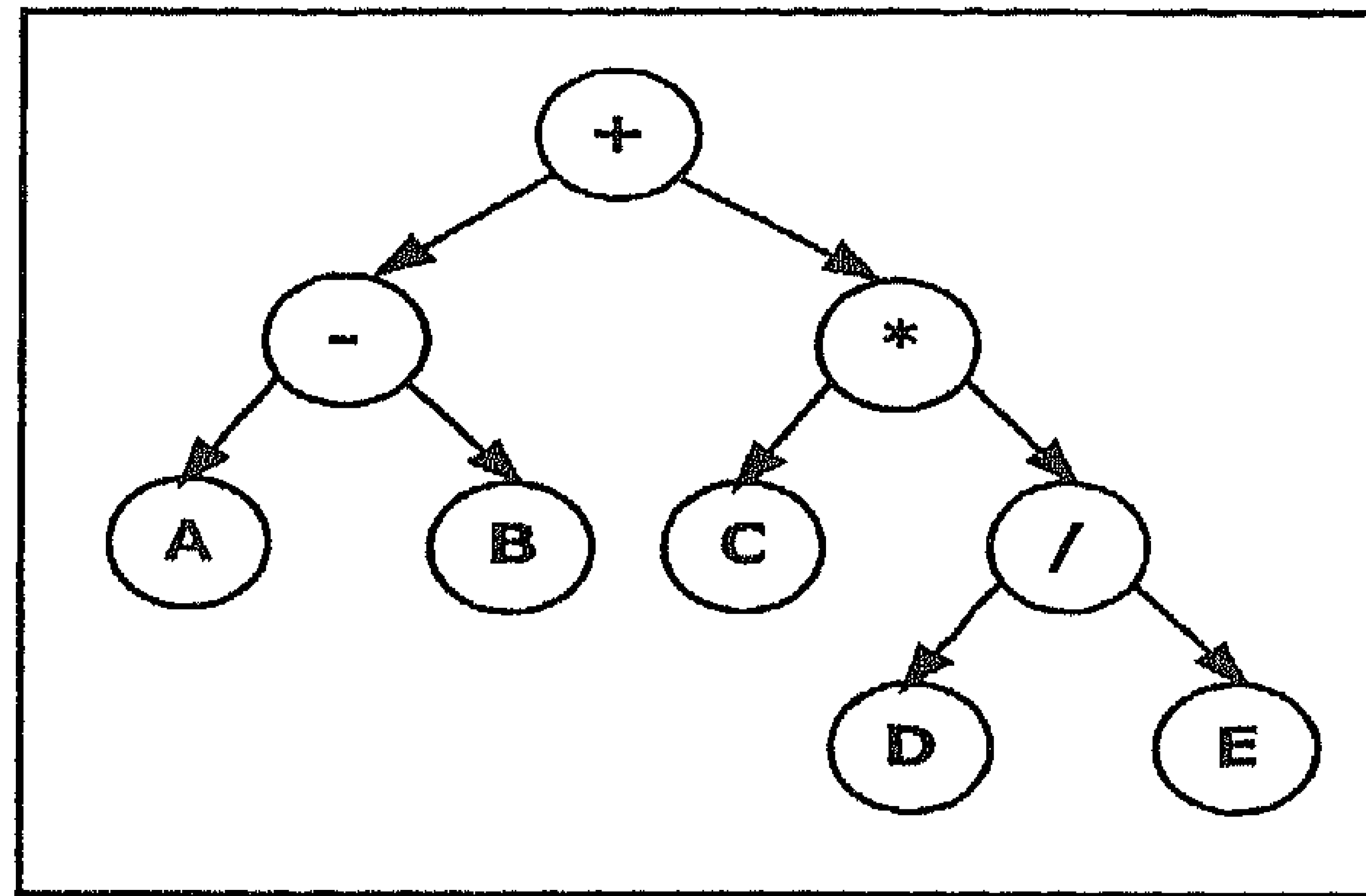
1. LDR (Inorder) التحرك باتجاه الابن الأيسر، ثم طباعة البيانات، ثم التحرك، باتجاه الابن الأيمن.

2. LRD (Postorder) التحرك باتجاه الابن الأيسر، ثم باتجاه الابن الأيمن، ثم طباعة البيانات.

3. DLR (Preorder) طباعة البيانات، ثم التحرك باتجاه الابن الأيسر، ثم باتجاه الابن الأيمن.

وتوجد علاقة بين طرق الاستعراض هذه والطرق الثلاث المختلفة لتمثيل التعبير الحسابي حيث أن طريقة الاستعراض LDR)) تنتج التعبير الحسابي الذي يحوي العمليات الحسابية بين المعاملات، وأن طريقة الاستعراض (LRD) تنتج في العادة تعبيراً حسابياً تقع العمليات الحسابية فيه بعد المعاملات، وأما طريقة الاستعراض الأخيرة (DLR) فينتج عنها تعبيراً حسابي تقع فيه العمليات الحسابية قبل المعاملات، وهذا ينتج طبعاً إذا كان الهيكل الشجري يمثل تعبيراً حسابياً.

ولتوضيح طرق الاستعراض المختلفة يمكننا الرجوع إلى الهيكل الشجري في الشكل (8) والذي نكرره هنا في الشكل (21).



الشكل (21)

يحتوي هذا الهيكل الشجري على تعبير حسابي يحتوي على العمليات الحسابية الثنائية (+، -، *، /) وعلى المعاملات (E, D, C, B, A). وفي الجزء التالي نبين طرق الاستعراض المختلفة على شكل إجراءات متكررة، ونبين كذلك نتيجة تطبيق هذه الطرق على الهيكل الشجري في الشكل (21).

أ- الاستعراض وفق السياق الوسطي LDR (Inorder Traversal):

يمكن تلخيص هذه الطريقة المتكررة كما يلي:

ابدأ من الجذر، وسر باتجاه اليسار (أي بأخذ اتجاه مؤشر الابن الأيسر) حتى لا تستطيع الذهاب أكثر وذلك بمصادفة القيمة NULL لمؤشر الابن الأيسر لعنصر من العناصر.

بعد ذلك اطبع القيمة الموجودة في ذلك العنصر ثم تحرك عنصراً واحداً إلى الوراء واطبع قيمة ذلك العنصر ثم تحرك باتجاه ابنه الأيمن إن وجد وكرر ما ذكر (أي المسير إلى اليسار) ابتداءً من هذا العنصر والذي يعامل معاملة الجذر.

وإذا لم يكن مستطاعاً المسير إلى اليمين (أي أن الابن الأيمن للعنصر الذي تم طباعة محتوياته يحمل القيمة NULL) تحرك إلى الوراء عنصراً آخر وكرر الخطوات المذكورة في بداية هذه الطريقة من طرق الاستعراض على الابن الأيمن. وفي حالة عدم وجود ابن أيمن كرر التراجع إلى الوراء عنصراً واحداً مع طباعة ذلك العنصر ثم الذهاب إلى الابن الأيمن ولحين إيجاد ابن أيمن، حيث يتم تكرار عملية المسير إلى اليسار وهكذا.

والدالة المتكررة التالية توضح هذه الطريقة:

```
void  treePtr::inOrder(treePtr* root)
 {
  if (root!=NULL)
   {inOrder(root->leftChild);
    cout<<root->info<<" ";
    inOrder(root->rightChild);
    }
  }
```

حيث يتضمن هذا الإجراء ثلاث خطوات رئيسة متكررة (Recursive) هي:

1 - زيارة الشجيرة اليسرى (Visiting Left Subtree).

2 - زيارة الجذر الذي يحوي البيانات أو المعلومات المرغوب عرضها.

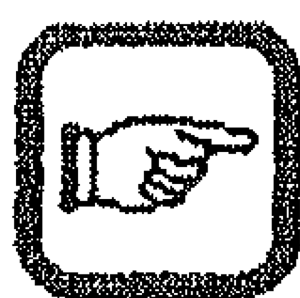
3 - زيارة الشجيرة اليمنى (Visiting Right Subtree).

وتنفذ هذه الخطوات، عزيزي الدارس، بأسلوب الاستدعاء الذاتي (Recursive).

وكما أشرنا فإن بساطة هذا الإجراء ينبع من كونه تكرارياً يعكس تعريف الهيكل الشجري، والذي أشرنا بأنه تكراري أيضاً.

ويمكن إعادة كتابة إجراء الاستعراض وفق السياق الوسطي أعلاه بأسلوب لا يستخدم الاستدعاء الذاتي (Non-recursive)، وهذا يتطلب جهداً أكثر. كما نحتاج إلى مكدس (Stack) وبعض المتغيرات الأخرى لتنفيذه كما هو مبين أدناه:

```
void treePtr::InOrder(treePtr* root)
 {//Nonrecursive version of Inorder procedure using a stack of Given size
  const stackSize = 200; //maximum number of Elements
  treePtr      *stack[stackSize];
  int  top;
  bool done;
  top= -1;       //initialize stack pointer named Top
  done=false;  //initialize Loop terminator
  do
   {while (root!=NULL)  //move down LeftChild fields
     {top++;
      if (top>stackSize)
       cout<<"stack is full, increase stacksize";
       else
        {stack[top]=root;
         root=root->leftChild;
        }
     }
    if (top!=0)
     {root=stack[top];
      top--;
      cout<<root->info<<" ";
      root=root->rightChild;
     }
     else
      done=true;
   }
  while (done);
 }
```



مثال (5)

عند تنفيذ طريقة الاستعراض وفق السياق الوسطي على الهيكل الشجري الوارد في الشكل (21)، يبدأ التنفيذ من الجذر (الذي يحتوي على إشارة الجمع) ونسير باتجاه الشجيرة اليسرى والتي تحتوي على عناصر ثلاث تحوي الرموز (-, A, B)، وبما أن العنصر الذي يحتوي على إشارة الطرح (-) هو جذر هذه الشجيرة نستمر بالمسير لحين الوصول إلى العنصر الذي يحوي الرمز (A)، حيث يتم طباعة (A) وذلك لكون الابن الأيسر لهذا العنصر يحتوي على القيمة (NULL).

بعد ذلك يتم الرجوع إلى العنصر الذي يحتوي على الرمز (-) ويتم طباعته. عندئذ نسير باتجاه الابن الأيمن للعنصر الذي يحتوي على إشارة السالب (-) ويتم طباعة (B)، بعدها يتم طباعة إشارة الجمع (+) ثم نسير باتجاه الابن الأيمن للجذر ونصل إلى العنصر الذي يحتوي على علامة الضرب (*) ونكرر عملية المسير إلى اليسار بالمسير باتجاه الابن الأيسر لهذا العنصر، حيث يتوقف المسير عند العنصر الذي يحتوي على القيمة C ويتم طباعتها.

بعد ذلك يتم التراجع إلى العنصر الذي يحتوي على علامة الضرب وذلك لعدم وجود ابن أيمن للعنصر C، ويتم طباعة إشارة الضرب(*).

ثم يتم الانتقال إلى العنصر الذي يحتوي على إشارة القسمة (/) ومن ثم يستمر المسير باتجاه الابن الأيسر لهذا العنصر حتى نجد مؤشراً قيمته تساوي (NULL). لذلك يتم طباعة الرمز (D) لعدم وجود ابن أيسر له وكذلك يتم التراجع إلى والد هذا العنصر لعدم وجود ابن أيمن له (الرمز D). لذلك يتم طباعة الرمز (/). وأخيراً نتجه باتجاه الابن الأيمن لهذا العنصر وينتهي استعراض الهيكل الشجري بطباعة الرمز (E). ونتيجة لهذا الاستعراض يتم طباعة العناصر حسب الترتيب التالي من اليسار إلى اليمين.

A - B + C * D / E

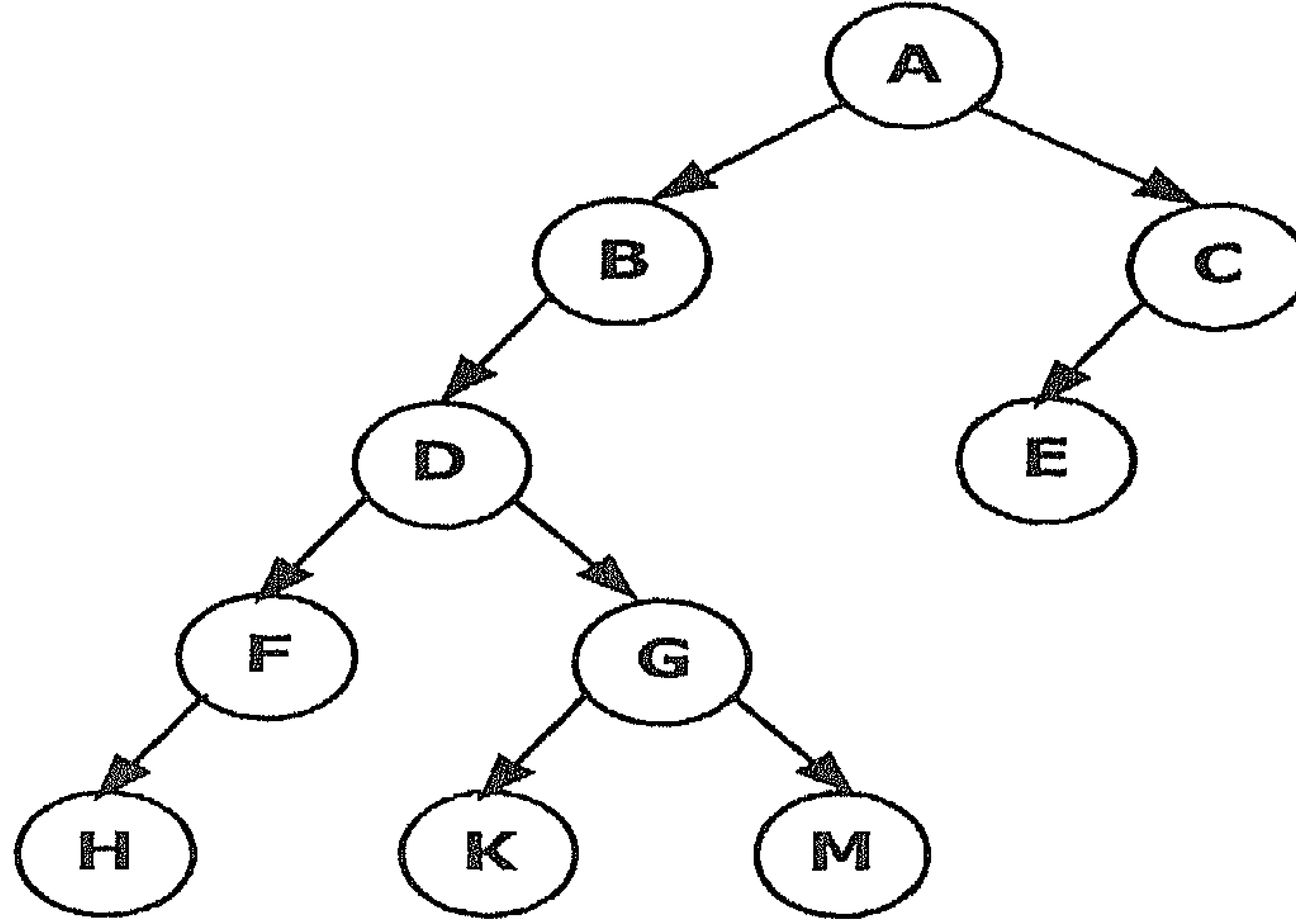
وهذا هو تعبير حسابي ويتبع نظام الرموز الوسطي (Infix Notation) أي أن العمليات الحسابية الثنائية تقع بين المعاملات.

وتوجد فائدة مهمة لهذا النوع من طرق الاستعراض وهي أنه عند تطبيقه على الهيكل الشجري الذي تم إنشاءه بالطريقة التي تم التعرف عليها عند بحث موضوع إنشاء الهيكل الشجري الثنائي في الجزء السابق فإن البيانات الناتجة تكون مرتبة ترتيباً تصاعدياً.

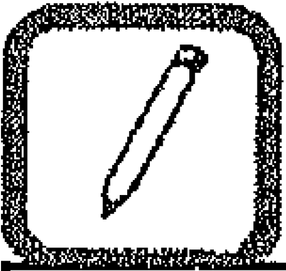


تدريب (3)

ما نتيجة الطباعة عند تنفيذ طريقة الاستعراض وفق السياق الوسطي على الهيكل الشجري المبين في الشكل (22):

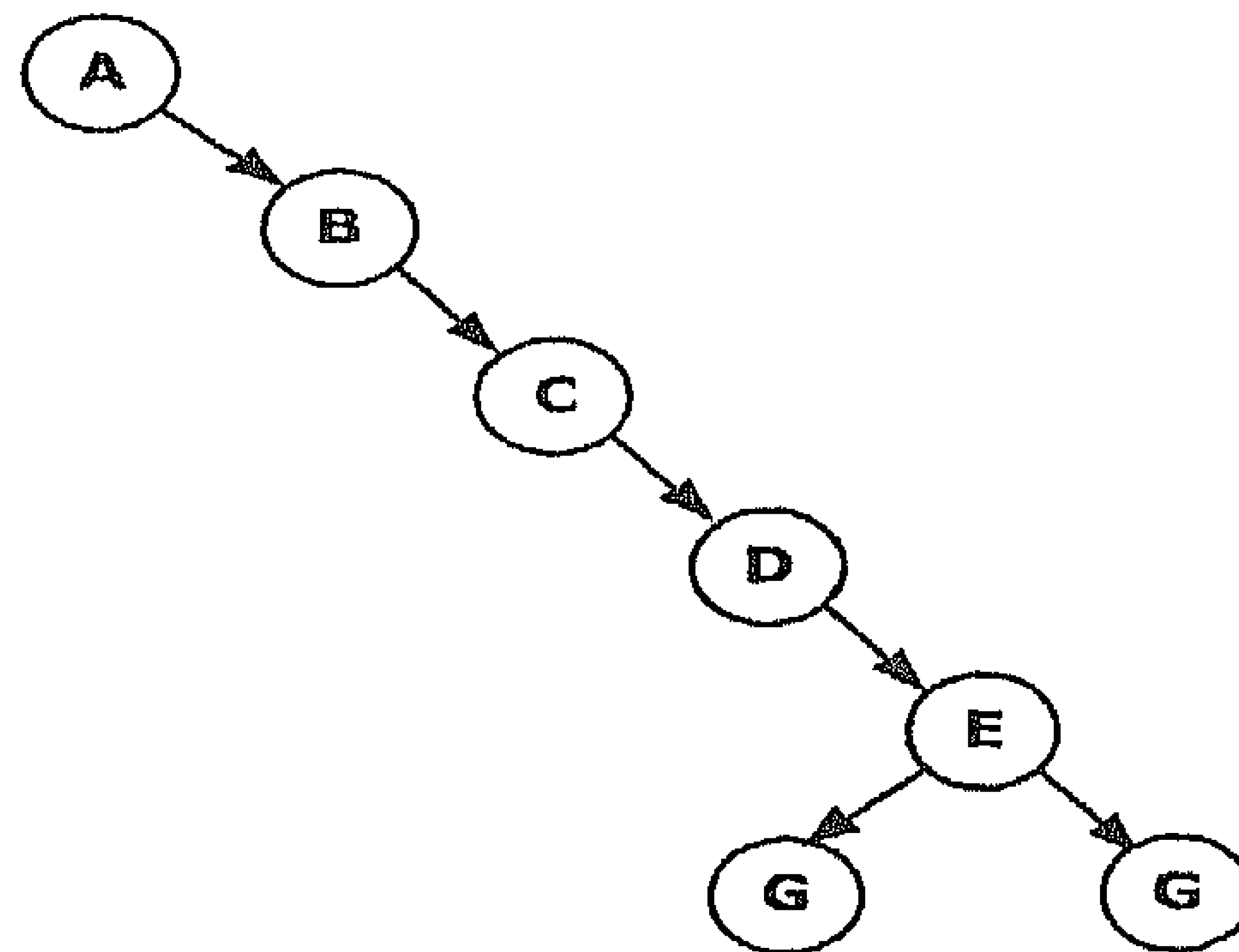


الشكل (22)



تدريب (4)

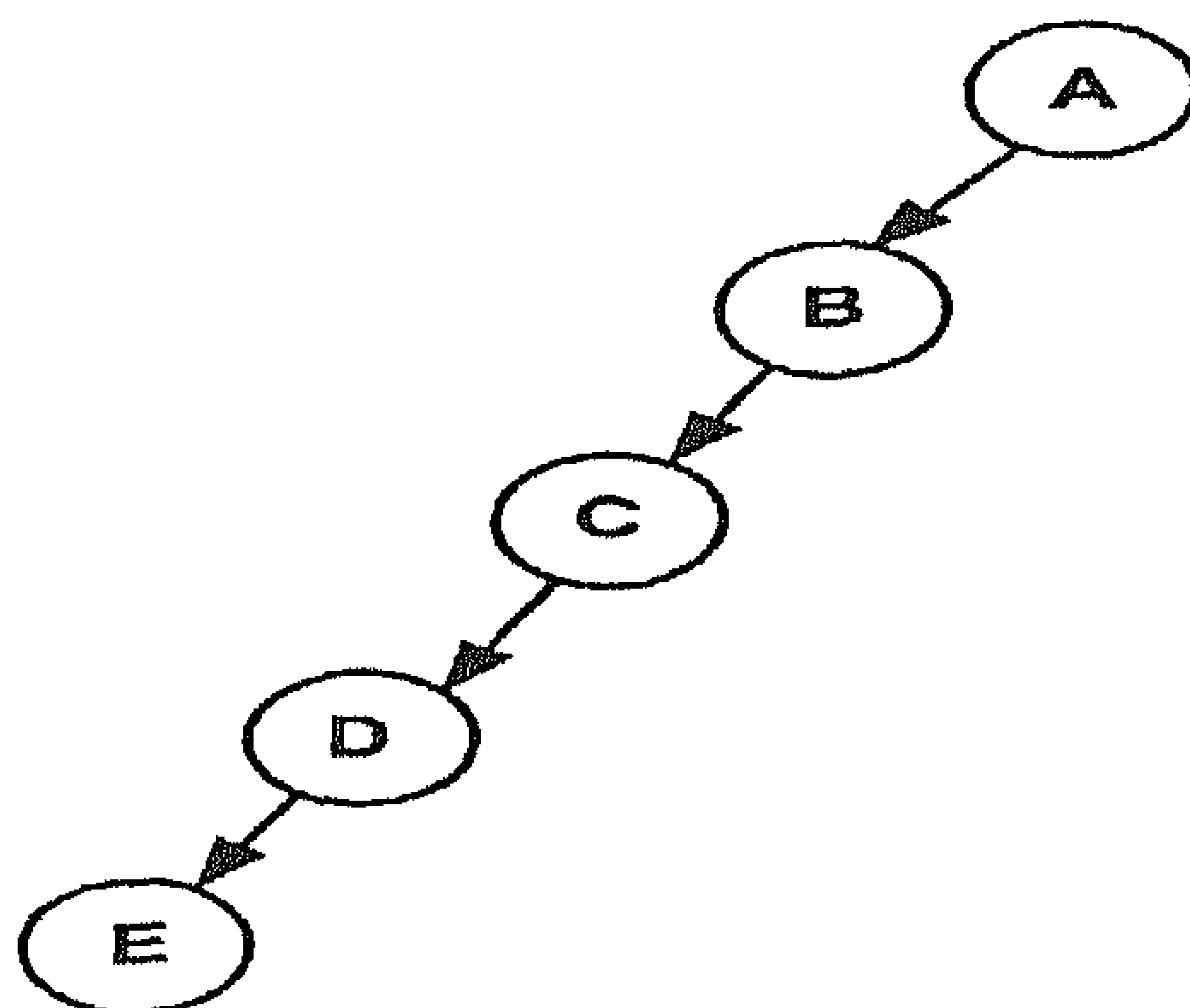
ما ترتيب استعراض عناصر الهيكل الشجري الثنائي المبين في الشكل (23) عند تطبيق طريقة الاستعراض وفق السياق الوسطي:



الشكل (23)

تدريب (5)

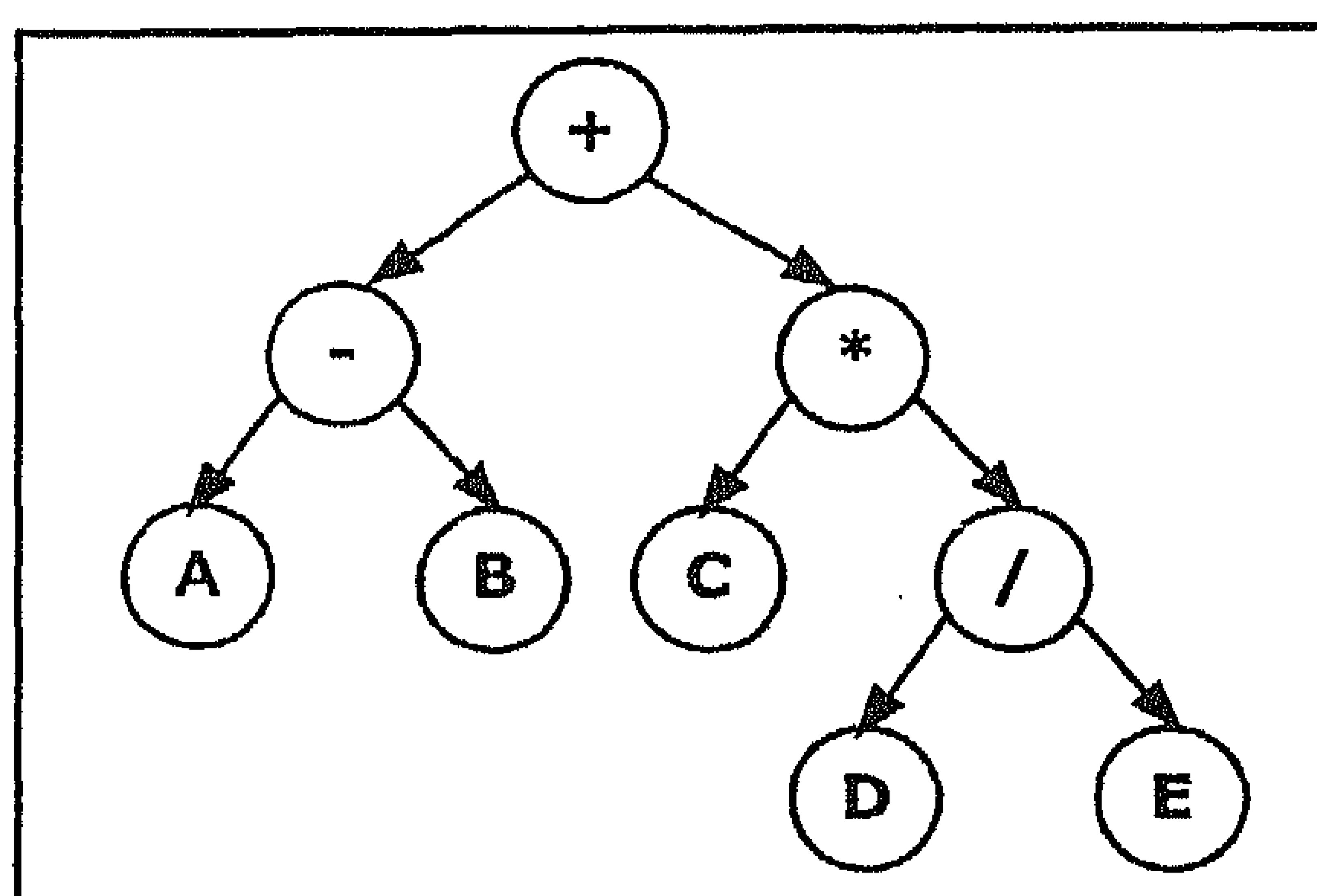
ما ترتيب استعراض عناصر الهيكل الشجري المبين في الشكل (24) عند تطبيق طريقة الاستعراض وفق السياق الوسطي:



الشكل (24)

ب- الاستعراض وفق السياق التبعي (PostOrder Traversal)

في هذه الطريقة لاستعراض عناصر الهيكل الشجري الثنائي، يتم استعراض الشجيرة اليسرى أولاً ثم الشجيرة اليمنى وأخيراً تتم زيارة محتويات العنصر أو طباعتها. وهذا يعني أنه لا تتم طباعة محتويات أي عنصرٍ إلاَّ بعد استعراض كل العناصر المكونة للشجيرة اليسرى والعناصر المكونة للشجيرة اليمنى له. فعلى سبيل المثال يمكننا تطبيق هذا المبدأ على الهيكل الشجري الوارد في الشكل (21) والذي نكرره هنا في الشكل (25).



الشكل (25)

وبناء عليه فإنه يتم زيارة العناصر على النحو التالي ومرتبة من اليسار إلى اليمين

A B – C D E / * +

وهذا يعني أن العمليات الحسابية تقع بعد المعاملات عند تطبيق هذه الطريقة من طرق الاستعراض على هيكل شجري ثنائي مكون لتعبير حسابي ويطلق على هذا الشكل من أشكال التعبير الحسابي "نظام الرموز التبعي" (PostOrder Notation). ونرى من هذا الشكل أنه تتم طباعة محتويات العنصر أو زيارة العناصربعد الانتهاء من الشجيرة اليسرى والشجيرة اليمنى له، أي أنه يتم استعراض العنصر عند مرور الخط المتقطع إلى يمينه وليس تحته كما كان الحال مع طريقة الاستعراض وفق السياق الوسطي.

وفي ما يلي،عزيزي الدارس، وصف للإجراء الدوري المتكرر الذي يستعرض عناصر الهيكل الشجري الثنائي وفق السياق التبعي:

```
void  treePtr::postOrder(treePtr* root)
   {if (root!=NULL)
{postOrder(root->leftChild);
postOrder(root->rightChild);
cout<<root->info<<" ";
}
   }
```

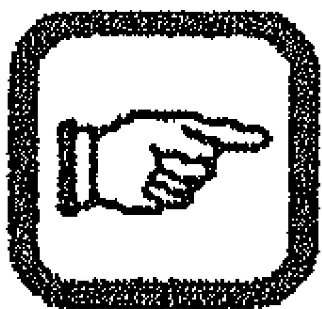
وكما نرى من هذا الإجراء فإنه يتضمن ثلاث خطوات رئيسة متكررة هي:

1 - زيارة الشجيرة اليسرى (Visiting Left Subtree).

2 - زيارة الشجيرة اليمنى (Visiting Right Subtree).

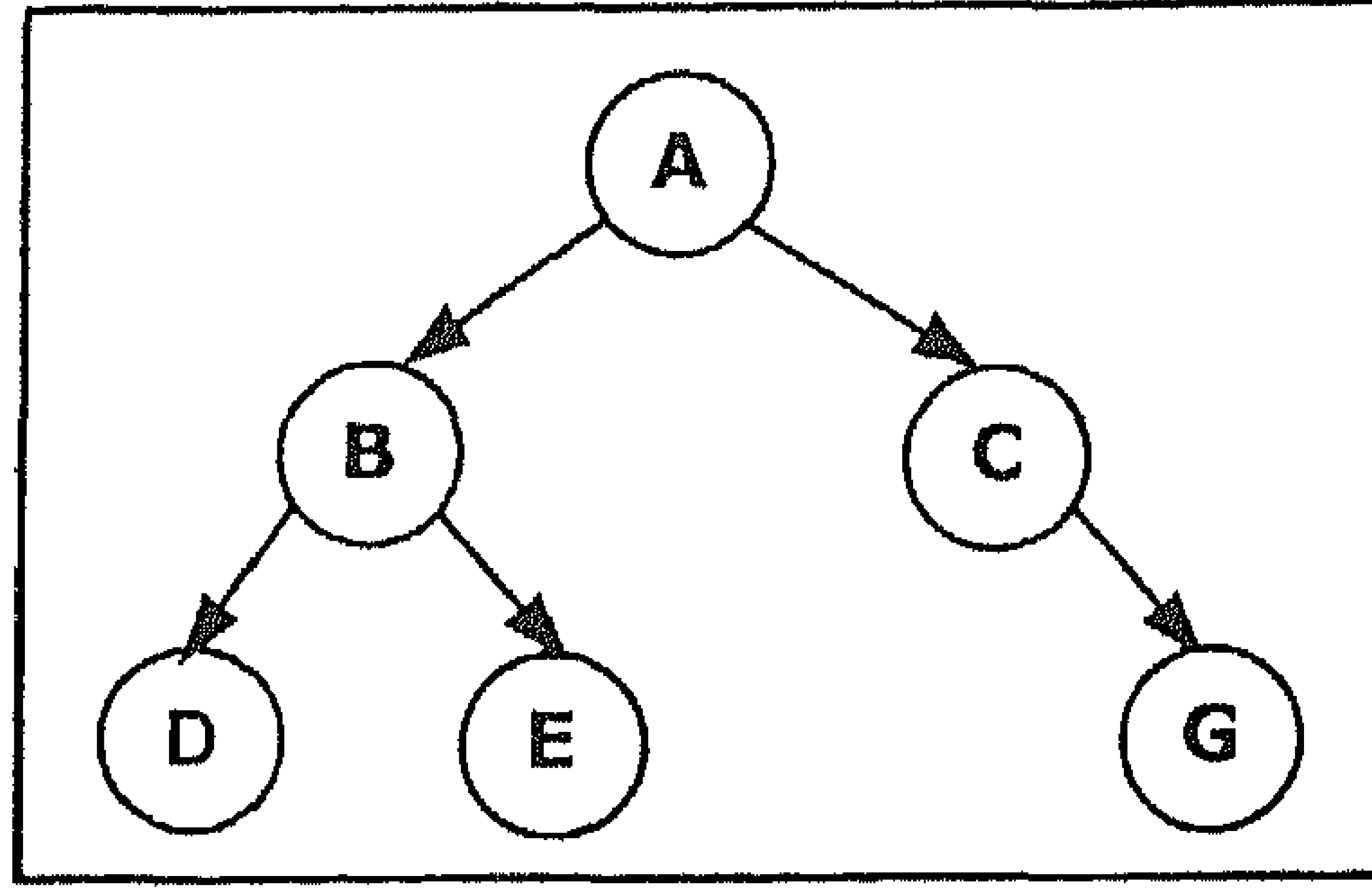
3 - زيارة العنصر الذي يحوي البيانات أو المعلومات المرغوب عرضها.

ويتم تنفيذ هذه الخطوات بأسلوب دوري متكرر (Recursive)، أي أنه لإتمام استعراض العنصر، يجب زيارة الابن الأيسر ثم الابن الأيمن له أولاً.



مثال (6)

لو فرضنا أن لدينا الهيكل الشجري الثنائي كما في الشكل (26)، فإنه بعد تنفيذ طريقة الاستعراض وفق السياق التبعي على هذا الهيكل الشجري ينتج عنه الترتيب التالي من اليسار إلى اليمين: D E B G C A

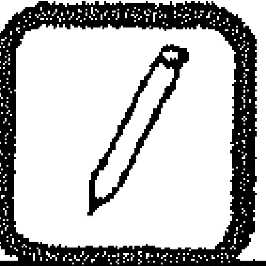


الشكل (26)



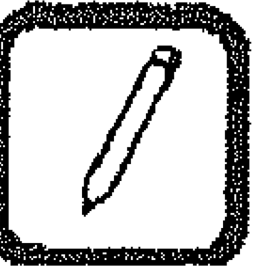
تدريب (6)

ما نتيجة الطباعة عند تنفيذ طريقة الاستعراض وفق السياق التبعي على الهيكل الشجري الوارد في الشكل (22) (تدريب (3)).



تدريب (7)

ما نتيجة الطباعة عند تنفيذ طريقة الاستعراض وفق السياق التبعي على الهيكل الشجري الوارد في الشكل (23)(تدريب (4)).



تدريب (8)

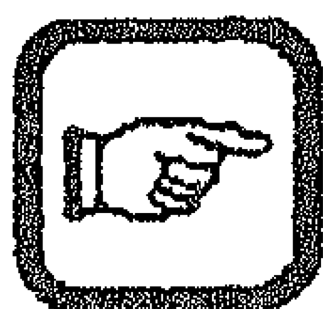
ما نتيجة الطباعة عند تنفيذ طريقة الاستعراض وفق السياق التبعي على الهيكل الشجري الوارد في الشكل (24) (تدريب (5)).

ج- الاستعراض وفق السياق القبْلي (PreOrder Traversal):

في هذه الطريقة لاستعراض عناصر الهيكل الشجري الثنائي، يتم طباعة (أو زيارة) محتويات العنصر أولاً، ثم زيارة الشجيرة اليسرى وأخيراً الشجيرة اليمنى. وهذا يعني أنه عند الوصول إلى عنصر من العناصر، يتم استعراض محتوياته أولاً ثم يستمر الانتقال إلى الشجيرات اليسرى لهذه العناصر حتى لا نستطيع التحرك أكثر من ذلك (أي لحين مصادفة القيمة NULL لمؤشر الابن الأيسر). بعد ذلك يتم التراجع عنصراً، ويعتبر هذا العنصر بمثابة الجذر الجديد ونكرر عليه ما تقدم.

والدالة الدورية المتكررة التالية، تبين كيفية تطبيق عملية استعراض عناصر الهيكل الشجري الثنائي وفق السياق القبلي:

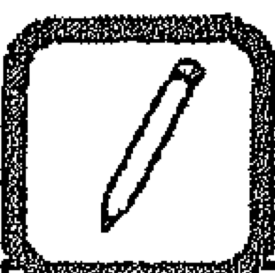
```
void  treePtr::preOrder(treePtr* root)
 {
  if (root!=NULL)
   {
    cout<<root->info<<" ";
    postOrder(root->leftChild);
    postOrder(root->rightChild);
   }
 }
```



مثال (7)

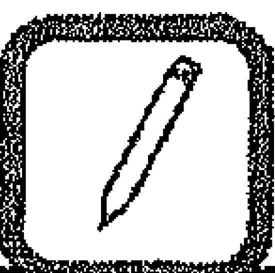
افرض أن لدينا الهيكل الشجري الثنائي في المثال السابق (الشكل (26))، فإن نتيجة الطباعة عند تنفيذ إجراء الاستعراض وفق السياق القبلي تكون كالآتي من اليسار إلى اليمين:

A B D E C G



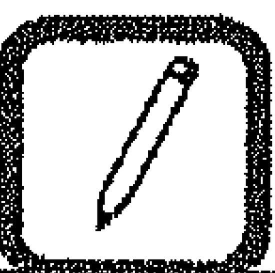
تدريب (9)

ما نتيجة الطباعة عند تنفيذ طريقة الاستعراض وفق السياق القبلي على الهيكل الشجري الوارد في الشكل (22) (تدريب (3)).



تدريب (10)

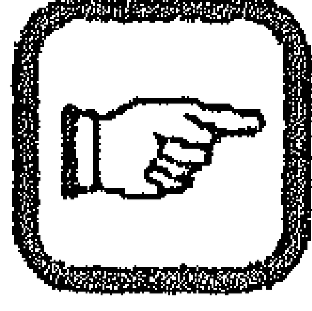
ما نتيجة الطباعة عند تنفيذ طريقة الاستعراض وفق السياق القبلي على الهيكل الشجري الوارد في الشكل (23) (تدريب (4)).



تدريب (11)

ما نتيجة الطباعة عند تنفيذ طريقة الاستعراض وفق السياق القبلي على الهيكل الشجري الوارد في الشكل (24) (تدريب (5)).

تجدر الإشارة أنه بالإمكان بناء الهيكل الشجري إذا عرفنا ترتيب العناصر من الاستعراض وفق السياق الوسطي وكذلك ترتيب العناصر من الاستعراض وفق السياق القبلي.



مثال (8)

افرض أن الاستعراض وفق السياق الوسطي للهيكل الشجري الثنائي المراد بناءه يُنتج الترتيب التالي للعناصر ومن اليسار إلى اليمين:

H D B G F E A C K

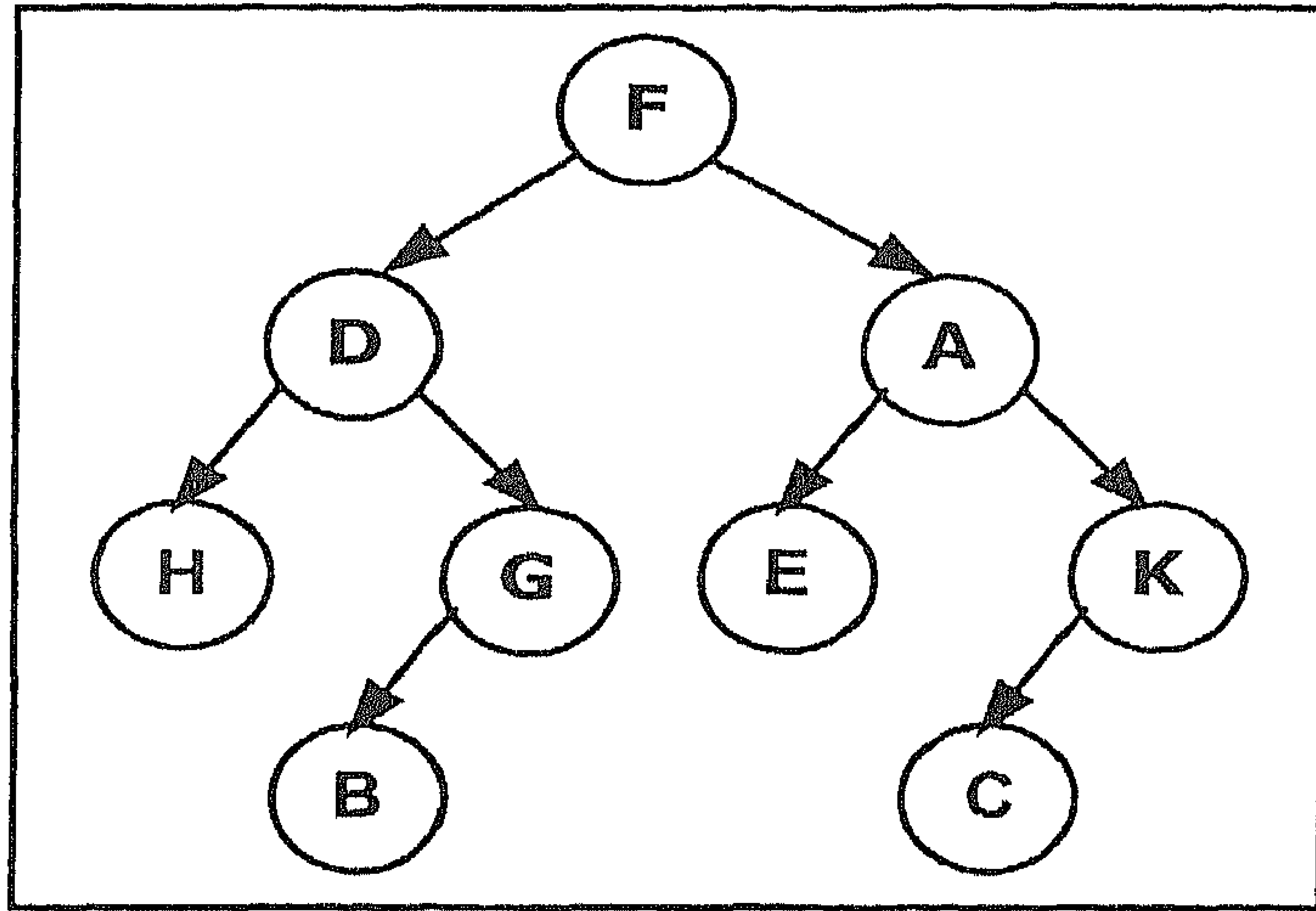
وأن الاستعراض وفق السياق القبلي للهيكل الشجري ينتج الترتيب التالي ومن اليسار إلى اليمين:

F D H G B A E K C

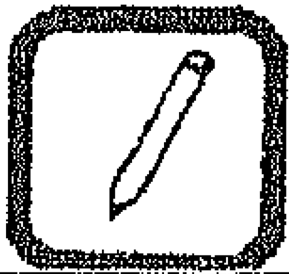
ارسم الهيكل الشجري.

الحل:

1. جذر الهيكل الشجري هو العنصر الأول من نتيجة الاستعراض وفق السياق القبلي، وهذا يحدد F على أنها الجذر.
2. بالرجوع إلى الاستعراض وفق السياق الوسطي، نستطيع تحديد الشجيرة اليسرى للجذر على أنها تتكون من جميع العناصر إلى يسار الجذر (F). أي أن الشجيرة اليسرى للجذر تتكون من العناصر (H, D, B, G). وكذلك فإن الشجيرة اليمنى للجذر (F) تتكون من العناصر الموجودة إلى يمين الجذر في الاستعراض الوسطي. وهذا يعني أن الشجيرة اليمنى للجذر تتكون من العناصر (E, A, C, K).
3. لتحديد الابن الأيسر للجذر (F) نرجع إلى الشجيرة اليسرى للجذر ونختار أول عنصر من قائمة هذه العناصر في الاستعراض وفق السياق القبلي وبما أن العنصر (D) هو أول هذه العناصر (من اليسار إلى اليمين) فإن العنصر (D) يمثل الابن الأيسر للجذر.
4. وبالأسلوب نفسه نحدد الابن الأيمن للجذر، وهذا يعطينا العنصر (A).
5. نكرر الطريقة ذاتها على العنصر (D) فنجد أن الشجيرة اليسرى له تتكون من العنصر(H)، وأن الشجيرة اليمنى لهذا العنصر تتكون من العناصر (B,G). ثم ننتقل إلى الاستعراض وفق السياق القبلي ونستنتج أن (G) تمثل الابن الأيمن للعنصر(D). وبالرجوع إلى الاستعراض وفق السياق الوسطي نجد أن العنصر (B) يقع إلى يسار العنصر (G) مما يدل على أن (B) يمثل الابن الأيسر للعنصر (G). وبتكرار ما تقدم على جميع العناصر نحصل على الهيكل الشجري المطلوب كما هو مبين في الشكل (27):



الشكل (27)



تدريب (12)

اقرض أن لديك هيكلاً شجرياً ثنائياً يتكون من عشرة عناصر وأن الاستعراض وفق السياق الوسطي هو:

A D E F G H J M R T

وأن الاستعراض وفق السياق القبلي هو:

J D A G E F H R M T

ارسم هذا الهيكل الشجري.

3.4 البحث عن عنصر معين في الهيكل الشجري الثنائي (Binary Tree Search)

من العمليات الممكن إجراؤها على الهياكل الشجرية بعد إنشائها، عملية الاستقصاء والبحث عن قيمة معينة أو سجل معين. وتُعتبر هذه العملية من العمليات الأساسية واللازمة قبل إجراء الإضافة إلى الهيكل الشجري أو الحذف منه، وذلك لإيجاد مواقع العناصر الضرورية لإجراء مثل هذه العمليات.

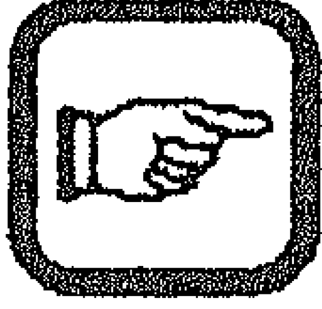
ومن الآن فصاعداً سوف نفرض أن الهيكل الشجري الثنائي قد أنشىء بالطريقة التي أشرنا إليها عند بحث موضوع إنشاء الهيكل الشجري الثنائي، أي أن الهيكل الشجري هو ثنائي البحث (Binary Search Tree).

وفي هذه الحالة، تعتبر عملية البحث عن قيمة معينة عملية سهلة. وقد تطرقنا إلى ذلك بنوع من التفصيل عند بحث موضوع إنشاء الهيكل الشجري الثنائي. وكما أسلفنا فإن معظم عمليات البحث عن موقع عنصر معين ربما تتطلب منا معرفة موقع ذلك العنصر (إذا كانت القيمة موجودة) وكذلك معرفة موقع والد العنصر. وفيما يلي الإجراء الذي يقوم بوظيفة البحث عن قيمة معينة في الهيكل الشجري ثنائي البحث.

```
treePtr* treePtr::searchBST(treePtr* root,int item,treePtr*
loc,treePtr* parent)
 {/*this procedure search a binary search tree,if item is in the
    tree it returns loc pointing to this node and parent points to its
parent if
   item is not intree,loc is equal to null */
  if (root==NULL)
   {loc=NULL;
    parent=NULL;
    }
    else
     {if (item==root->info)
       {loc=root;
        parent=NULL;
       }
       else
        {if (item<root->info)
          {loc=root->leftChild;
           parent=root;
          }
          else
           {loc=root->rightChild;
            parent=root;
            }
        while(loc!=NULL && item!=loc->info)
         {if(item<loc->info)
           {parent=loc;
            loc=loc->leftChild;
            }
            else
             {parent=loc;
              loc=loc->rightChild;
             }
         }
        }
     }
 }
```

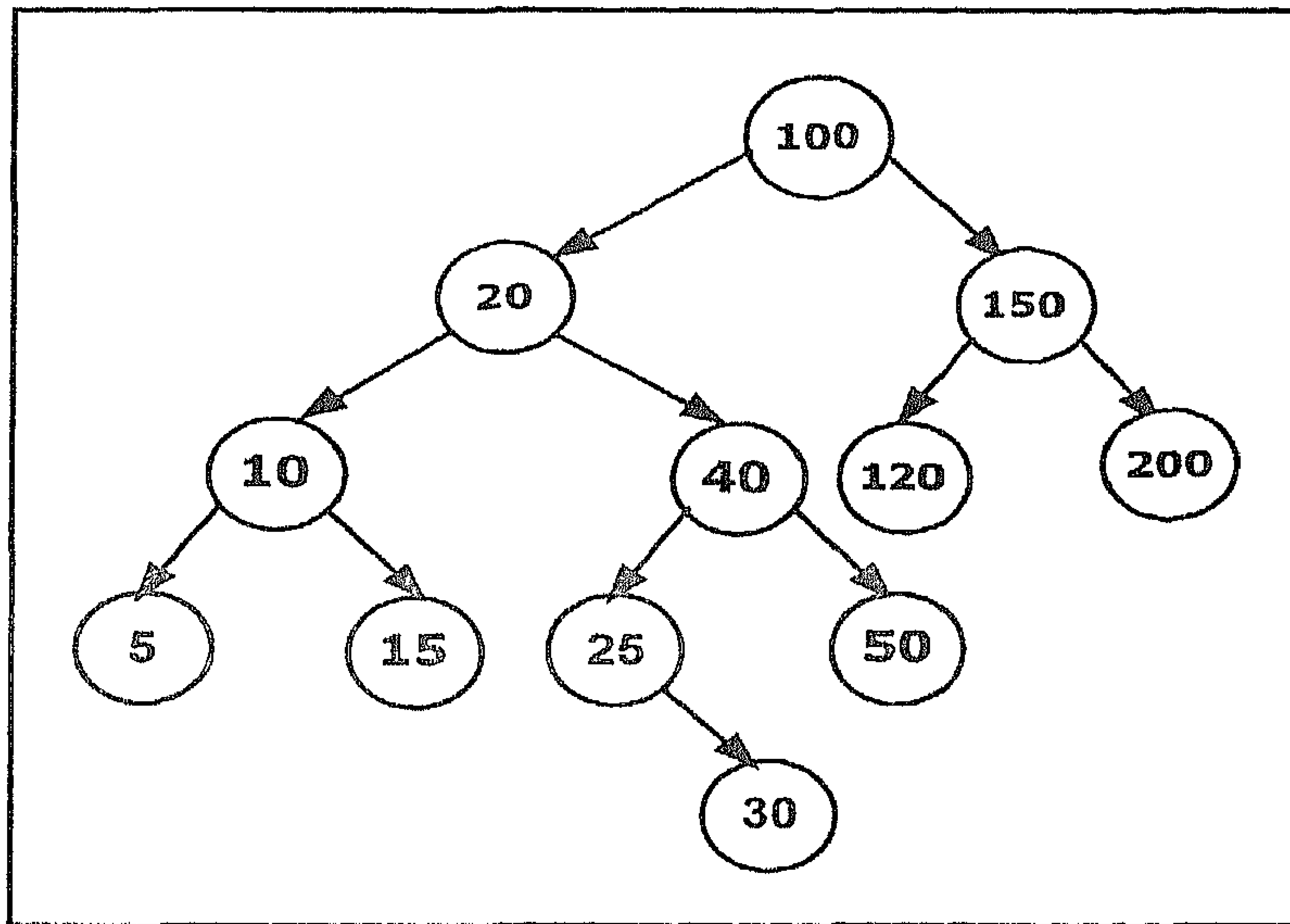
تلاحظ، عزيزي الدارس، أن الدالة تعتمد في خياراتها على قيم رقمية كما هو مبين من نوع العناصر في ترويسة الدالة (int item). ويمكن لهذه القيم أن تكون رمزية وذلك باستبدال النوع العددي الصحيح (int) بالنوع الرمزي في لغة سي++ (char). وكذلك يمكن للعنصر أن يكون سلسلة رمزية أو سجلاً أو أي نوع آخر يتم الإعلان عنه حسب الأصول.

ويمكن توضيح عمل هذا الإجراء بالمثال التالي:



مثال (9)

افرض أنك أعطيت الهيكل الشجري المبين في الشكل (28) وأنك تود البحث عن العنصر الذي يحتوي على القيمة: "30"



الشكل (28)

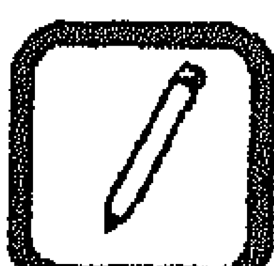
فعند تطبيق الإجراء (SearchBST) السابق على هذا الهيكل الشجري نحصل على الخطوات التالية:

1. ابدأ بالمقارنة من الجذر حيث تقارن القيمة "30" مع القيمة "100"، وبما أن القيمة "30" أصغر من القيمة "100" تتحرك إلى الابن الأيسر للجذر أي العنصر الذي يحتوي على القيمة "20".
2. قارن القيمة 30 مع القيمة 20، وبما أن القيمة 30 أكبر من القيمة 20 يتم التحرك إلى الابن الأيمن لهذا العنصر (أي إلى العنصر الذي يحتوي القيمة 40).

3. قارن القيمة 30 مع القيمة 40. وبما أن القيمة 30 أصغر من القيمة 40 يتم التحرك إلى الابن الأيسر لهذا العنصر (أي إلى العنصر الذي يحتوي على القيمة 25).
4. قارن القيمة 30 مع القيمة 25. وبما أن القيمة 30 أكبر من 25 يتم التحرك إلى الابن الأيمن لهذا العنصر (أي العنصر الذي يحتوي على القيمة 30).
5. وأخيراً فإن المؤشر (Loc) سوف يؤشر على هذا العنصر وأن (parent) سيؤشر على العنصر الذي يحتوي على القيمة 25. وعندها ينتهي هذا الإجراء.

4.4 الإضافة إلى الهيكل الشجري الثنائي (Binary Search Tree Insertion)

لقد تطرقنا إلى عملية إضافة عنصر جديد إلى الهيكل الشجري عند بحث موضـوع إنشاء الهيكل الشجـري. فيمكنك، عزيزي الدارس، استخدام الإجـراء نفسه الوارد هناك والذي أسميناه (insertNode)، لإضافة عناصر جديدة إلى الهيكل الشجري وذلك بتمرير هذه القيمة الجديدة كمعامل حقيقي إلى الإجراء والذي يقوم بدوره بعملية الإضافة.



تدريب (13)

افرض أنك أعطيت مجموعة من الأحرف وترغب إدخالها في هيكل شجري ثنائي البحث وفق الترتيب التالي ومن اليسار إلى اليمين: J, R, D, G, T, E, M, H, A, F ارسم هذا الهيكل الشجري بحيث يعطيك استعراض العناصر وفق السياق الوسطي يعطيك العناصر مرتبة ترتيباً تصاعدياً.

5.4 الحذف من الهيكل الشجري الثنائي (Binary Search Tree Deletion)

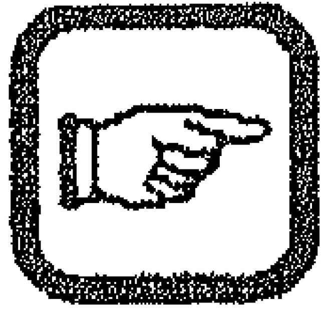
بعد عملية الإضافة، نتحول إلى عملية حذف عنصر من الهيكل الشجري. وللتأكد من وجود العنصر المراد حذفه في الهيكل الشجري، لا بد من تنفيذ عملية البحث والمتمثلة بالدالة (searchBST) أولاً، ومن ثم يتم إجراء عملية الحذف وسنطلق على هذه الدالة اسم (deleteNode).

ويمكن الحصول على الحالات التالية بعد تنفيذ عملية البحث عن العنصر وذلك تمهيداً لإجراء عملية الحذف.

1. العنصر الذي تم البحث عنه والمراد حذفه غير موجود.
2. عدم وجود أبناء للعنصر المراد حذفه.
3. وجود ابن واحد فقط للعنصر المراد حذفه.
4. وجود ابنين للعنصر المراد حذفه.

ففي الحالة الأولى (أي عدم وجود العنصر المراد حذفه في الهيكل الشجري) نكتفي بطباعة إعلان يخبر بأن ذلك العنصر غير موجود في الهيكل الشجري.

أما في الحالة الثانية أي في حالة عدم وجود أبناء للعنصر المراد حذفه فإن كل ما نريد عمله هو تغيير مؤشر الوالد الذي يشير إلى العنصر لكي تصبح قيمته مساوية لـِ NULL. وهذا يعني أن العنصر المطلوب حذفه لم يعد موجوداً في الهيكل الشجري، ومن ثم يتم إعادة العنصر المحذوف إلى ذاكرة الحاسوب المتوفرة للاستعمال، والمثال التالي يوضح ذلك.

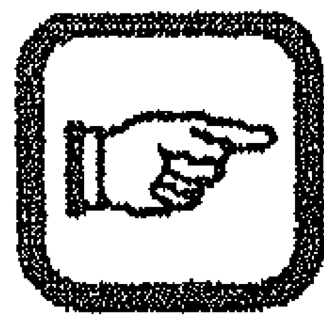


مثال (10)

افرض أنك تريد حذف العنصر الذي يحتوي على القيمة 200 من الهيكل الشجري في المثال (9).

ففي هذه الحالة، وبما أنه لا يوجد أبناء لهذا العنصر، فإن كل ما هو مطلوب هو إعادة العنصر الذي يحتوي على القيمة 200 إلى ذاكرة الحاسوب المتوفرة للاستعمال وذلك عن طريق جملة سي++ (delete)، ومن ثم جعل قيمة مؤشر الابن الأيمن للعنصر الذي يحتوي على القيمة 150 (أي والد العنصر المراد حذفه) مساوية للقيمة (NULL).

وأما في حالة وجود ابن واحد للعنصر المراد حذفه فيمكن تلخيص ما يلزم عمله كما في المثال التالي:



مثال (11)

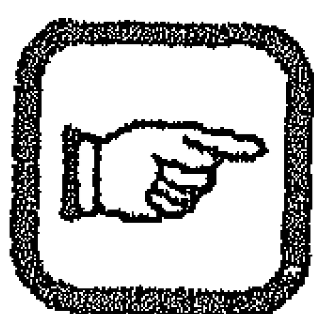
افرض أنك تريد حذف العنصر الذي يحتوي على القيمة 25 من الهيكل الشجري في المثال (9).

يلزمك، عزيزي الدارس، في هذه الحالة تعديل قيمة المؤشر المنبثق من والد العنصر المراد حذفه إلى العنصر ليؤشر على ابن هذا العنصر الوحيد (أي العنصر الذي يحتوي القيمة 30)، وهذا يعني أن قيمة مؤشر الابن الأيسر للعنصر الذي يحتوي على القيمة 40 ستعدل بحيث سيؤشر مؤشر الابن الأيسر على العنصر الذي يحوي القيمة 30، ومن ثم يعاد هذا العنصر إلى ذاكرة الحاسوب (العنصر الذي قيمته 25) لاستعماله في عمليات الإضافة المستقبلية.

أما الحالة الأخيرة من حالات الحذف فهي الأكثر تعقيداً من الحالات السابقة. وهي تتم كما يلي:

1. استخدام الهيكل الشجري لكتابة العناصر وفق السياق الوسطي.
2. سمِّ العنصر المراد حذفه N .
3. جد القيمة S(N)، هذه القيمة تمثل العنصر التابع المباشر (Inorder Successor) للعنصر المراد حذفه.
4. قم بحذف العنصر S(N) من الهيكل الشجري المعطى.
5. استبدل العنصر المراد حذفه N بالعنصر S (N) الذي كنت قد حذفته.

ويمكن توضيح ذلك بالمثال التالي:



مثال (12)

بالرجوع إلى الهيكل الشجري في المثال (9)، افرض أنك تريد حذف العنصر الذي يحتوي على القيمة 20 حيث أن هذا العنصر له ابنان اثنان.

أولاً: ترتيب العناصر وفق السياق الوسطي هو:

200 150 120 100 50 40 30 25 20 15 10 5

ومن هذا الترتيب نجد أن S(20) هو العنصر الذي يحتوي على القيمة 25. تبدأ عملية الحذف بحذف العنصر الذي يحتوي على القيمة 25 أولاً، ومن ثم استبدال العنصر المراد حذفه (والذي يحتوي على القيمة 20 بالعنصر S(20) والذي يحتوي على القيمة 25، وتتم هذه العملية بإجراء التعديل على مؤشرات الربط فقط وليس بتحريك البيانات والقيم التي تحتوي عليها العناصر.

ويمكن كتابة الإجراءات اللازمة لحذف عنصر من عناصر الهيكل الشجري، والذي يغطي الحالات السابقة على الشكل المبين أدناه باستخدام لغة سي++.

تذكر، عزيزي الدارس، أن تنفيذ هذه الإجراءات يتم بعد استدعاء دالة البحث عن العنصر المراد حذفه. وكنا قد أشرنا إلى هذه الدالة باسم (searchBST)، حيث تعيد هذه الدالة موقع

العنصر المراد حذفه إن كان موجوداً ويشير إلى هذا الموقع المؤشر (loc). وكذلك فإن المؤشر (parent) يشير إلى والد هذا العنصر. أما إذا لم يكن العنصر موجوداً فيعيد هذا الإجراء القيمة (NULL) للمؤشر (loc). وإذا كان العنصر المراد حذفه يمثل جذر الشجرة فإن قيمة (parent) تصبح (NULL).

```
treePtr*  treePtr::deleteNode (treePtr* root,int item)
  {treePtr *loc,*parent;
          searchBST(root, item, loc, parent);
   if(loc==NULL)
    cout<<"item is not in tree"<<endl;
    else
     {if (loc->rightChild !=NULL && loc->leftChild!=NULL)
          case4(root,loc, parent);
      else
         cases1_2_3(root, loc, parent);
       delete(loc);
     }
  }

////////////////////////////////////////////////

treePtr*  treePtr::cases1_2_3(treePtr* root,
                  treePtr* loc,treePtr* parent)
  {treePtr* child;
   if(loc->leftChild==NULL && loc->rightChild==NULL)
    child=NULL;
    else
     if (loc->leftChild !=NULL)
      child=loc->leftChild;
      else
       child=loc->rightChild;
   if (parent!=NULL)
    {if (loc=parent->leftChild)
      parent->leftChild=child;
      else
       parent->rightChild=child;
    }
    else
     root=child;
   return root;
  }
```

```
////////////////////////////////////////
treePtr* treePtr::case4 (treePtr* root,treePtr* loc,treePtr* parent)
 {treePtr *succ, *succparent;
  //find inorder successor,succ,and parent of inorder successor succparent
  succ=loc->rightChild;
  succparent=loc;
  while (succ->leftChild!=NULL)
    {succparent=succ;
     succ=succ->leftChild;
    }
  cases1_2_3(root, succ, succparent);
  //now replace node to be deleted with its inorder successor
  if (parent !=NULL)
   {if(loc ==parent->leftChild)
     parent->leftChild= succ;
     else
      parent->rightChild=succ;
    }
    else
     root=succ;
     succ->leftChild=loc->leftChild;
     succ->rightChild=loc->rightChild;
 }
```

حيث تعالج الدالة (Cases1_2_3) الحالات الثلاث الأولى من حالات حذف العناصر من الهيكل الشجري لأن العنصر لا يحتوي على ابنين. وفي هذه الدالة فإن المؤشر (parent) يؤشر على موقع العنصر المراد حذفه. أما إذا كان العنصر المراد حذفه يمثل الجذر فإن قيمة هذا المؤشر تصبح (NULL). أما مؤشر الربط (Child)، فقد استعمل للتأشير على الابن الوحيد للعنصر المراد حذفه إذا كان له ابن، وإلاّ فإن قيمته تساوي (NULL).

أما الإجراء (Case4) فيستعمل لحذف عنصر له ابنان اثنان. ويشير المؤشر (succ) إلى موقع العنصر التابع للعنصر المراد حذفه عند استعراض العناصر وفق السياق الوسطي. أما المؤشر (succParent) فيشير إلى موقع والد العنصر التابع للعنصر المراد حذفه وفق السياق الوسطي.

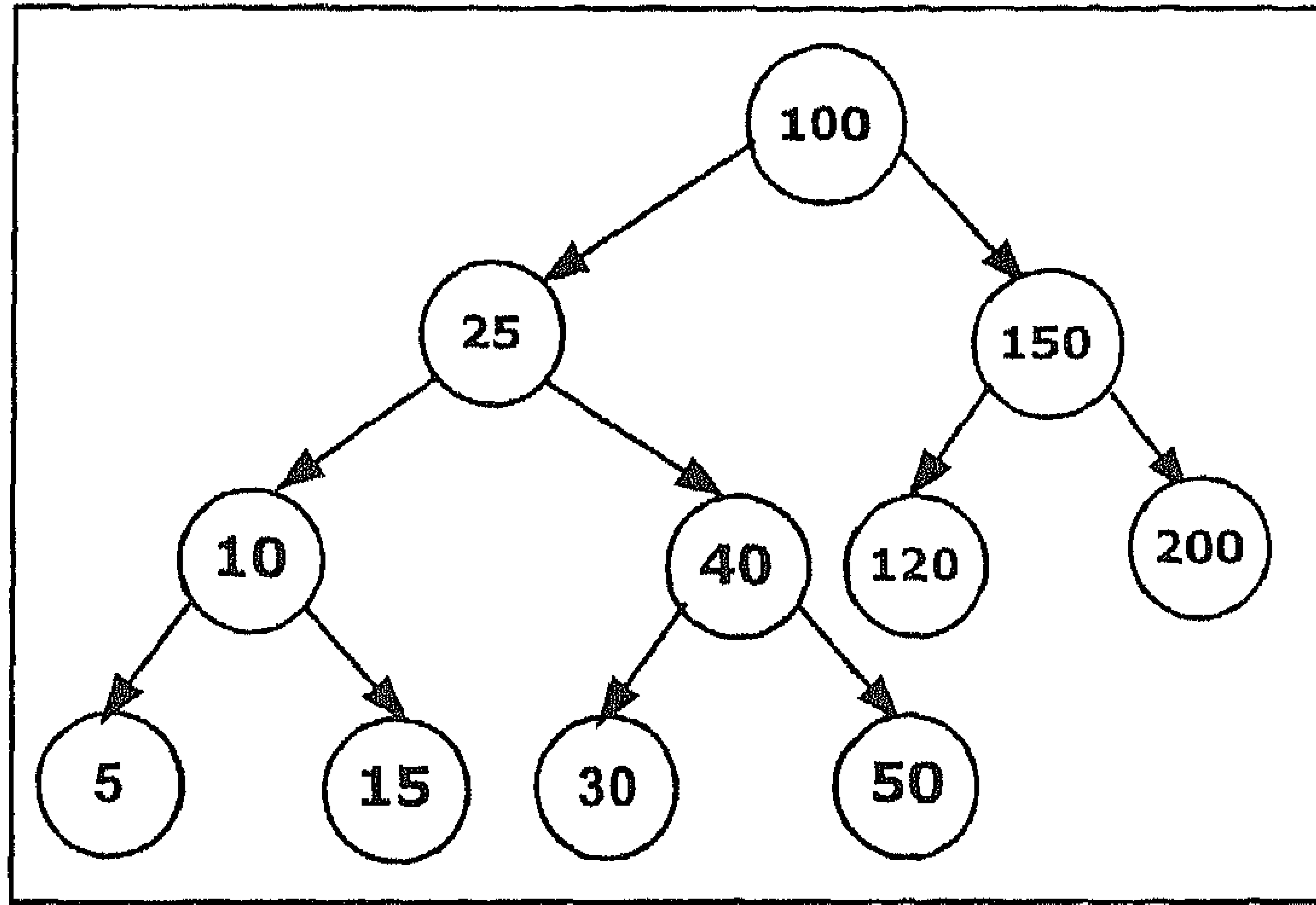
ويمكننا تطبيق هذه الإجراءات كما وردت في مثال (8) كما يلي:

مثال (13)

عد إلى الشكل (28) الوارد في مثال (9) وارسم الهيكل الشجري بعد حذف العنصر الذي يحتوي على القيمة 20.

الحل:

بعد حذف العنصر الذي يحتوي على القيمة 20 نحصل على الهيكل الشجري التالي (الشكل (29)).



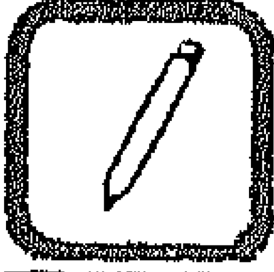
الشكل (29)

وتفسير ذلك كما يلي:

تلاحظ أن العنصر الذي يحتوي على القيمة 20 قد حذف من الهيكل الشجري كما في الشكل (29)، ولذلك فإن هذا المثال يعتبر تطبيقاً على الحالة الرابعة من حالات حذف عنصر من عناصر الهيكل الشجري. لذلك تتمثل الخطوة الأولى من خطوات الحل في تحديد العنصر التابع مباشرة للعنصر المراد حذفه عند تطبيق طريقة الاستعراض وفق السياق الوسطي. ونرمز لهذا العنصر بالرمز S(N). وعند تطبيق طريقة الاستعراض وفق السياق الوسطي على الهيكل الشجري نحصل على الترتيب التالي للعناصر:

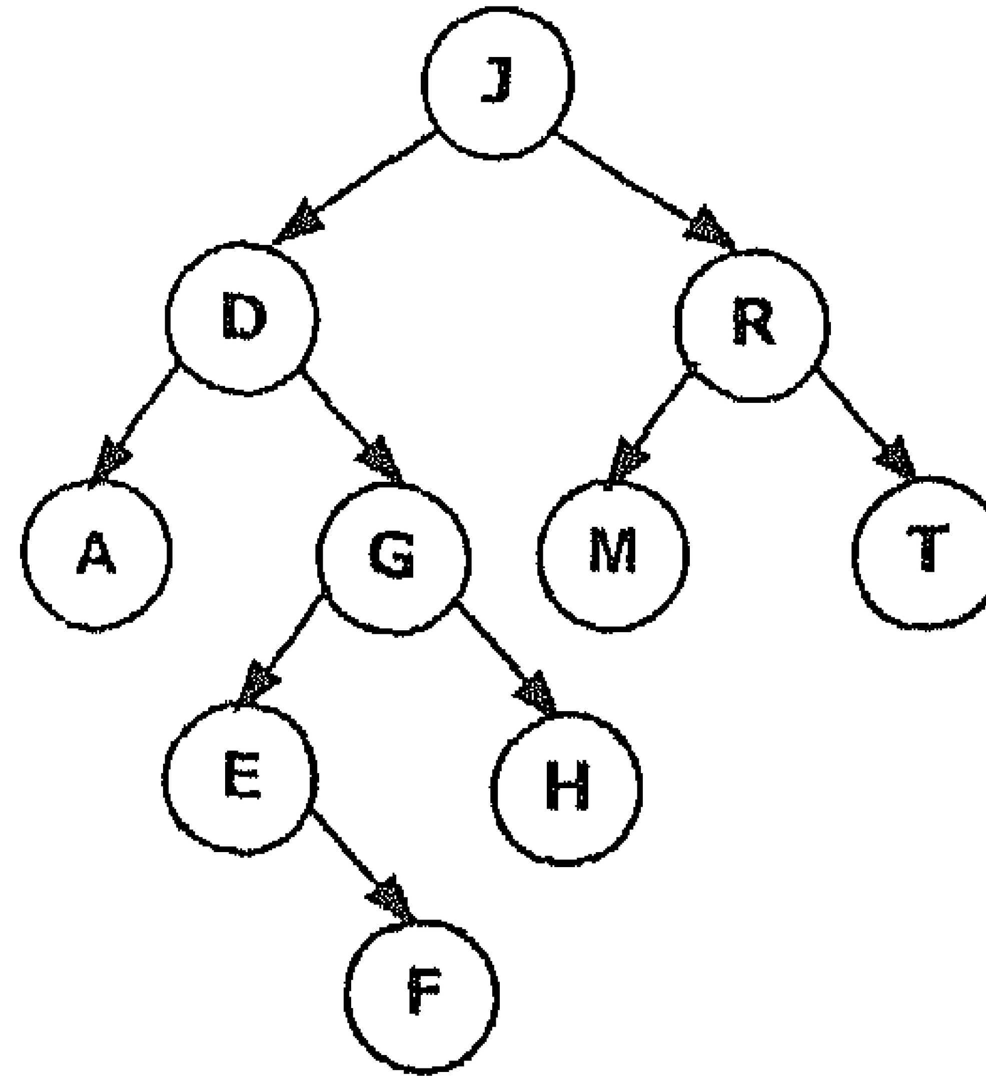
5 10 15 20 25 30 40 50 100 120 150 200

ولذلك فإن العنصر الذي يحتوي على القيمة 25 هو التابع للعنصر الذي يحتوي على القيمة 20 عند تطبيق طريقة الاستقصاء وفق السياق الوسطي. ونتيجة لذلك فإنه يتم استبدال القيمة 20 بالقيمة 25. بعد ذلك يصبح العنصر الذي يحتوي على القيمة 30 ابناً أيسراً للعنصر الذي يحتوي على القيمة 40 وذلك للمحافظة على خاصية الهيكل الشجري ثنائي البحث.



تدريب (14)

افرض أنك أعطيت الهيكل الشجري التالي (الشكل 30):



الشكل (30)

ارسم الهيكل الشجري بعد:

(أ) حذف العنصر G.

(ب) حذف العنصر E من الهيكل الجديد.

6.4 ترتيب القيم المخزنة في الهيكل الشجري الثنائي (Binary Tree Sort)

كما أسلفنا يمكن القيام بمثل هذه العملية باستخدام إجراء إنشاء الهيكل الشجري لإنشاء هيكل شجري ثنائي البحث (الاستقصاء) بحيث تكون البيانات مرتبة وفق السياق الوسطي عند تنفيذ طريقة الاستعراض عليها.

ويمكن بيان برنامج ترتيب قيم الهيكل الشجري كما يلي:

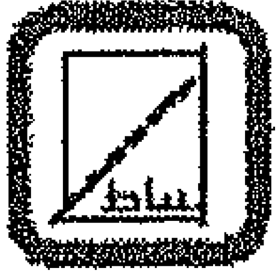
```
class treePtr
{
        treePtr * leftChild;
        treePtr * rightChild;
        int info,ch;
public:
  treePtr*  treeInit(treePtr*);
  treePtr*  insertNode(treePtr*,int val);
  void     inOrder(treePtr*);
};
```

```
void main()
 {treePtr *root=NULL,d1;
  char val;
  fstream inFile("c:\\BSTSort.txt",ios::in);
  if (!inFile.eof())
   {root=d1.treeInit(root);
    while (! inFile.eof())
     {inFile>> val;
      root=d1.insertNode(root,val);
      }
    d1.inOrder(root);
    }
  }
```

لاحظ، عزيزي الدارس، أن عناصر الهيكل الشجري تحتوي على قيم عددية، مع العلم أنها يمكن أن تحتوي على بيانات من أنواع مختلفة، ويمكن كذلك أن تحتوي على سجلات. وفي هذه الحالة يتم ترتيب العناصر حسب مفتاح معين في السجل.

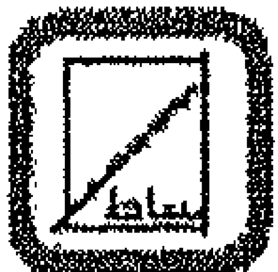
بالإضافة إلى العمليات الست الأساسية الآنفة الذكر، توجد عمليات غير مستعملة كثيراً من ضمنها:

- عمل نسخة من الهيكل الشجري الثنائي (Copy a Binary Tree).
- مقارنة ما إذا كان هيكل شجري (A) نسخة طبق الأصل من الهيكل الشجري (B).



نشاط (1)

أجر التعديل اللازم على الإجراء (insertNode) في برنامج إنشاء الهيكل الشجري الثنائي في البند (ج) التابع للقسم (1.4) بحيث يتم طباعة الأعداد بشكل تنازلي عند تطبيق طريقة الاستعراض وفق السياق الوسطي.



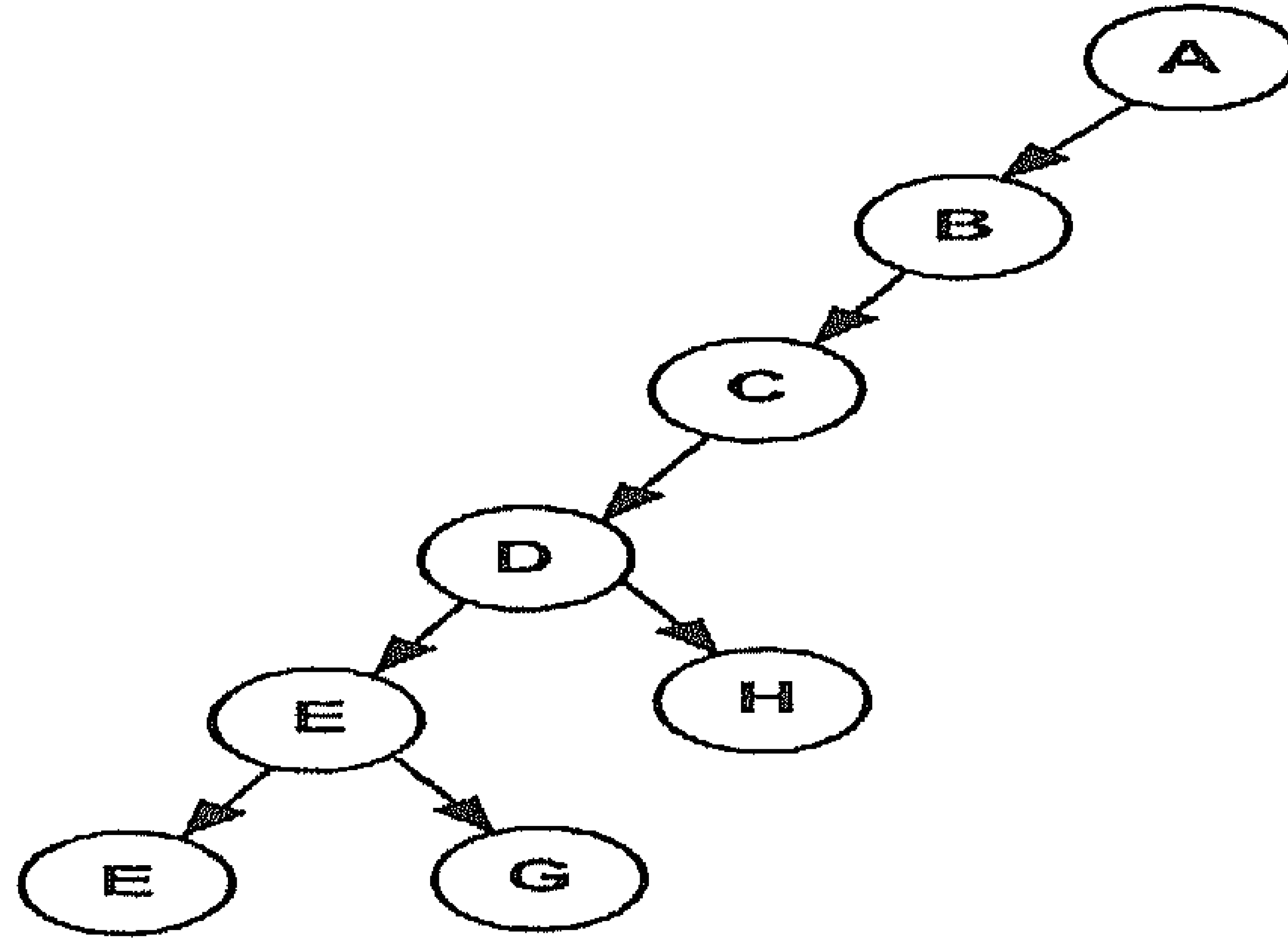
نشاط (2)

افرض أن لديك ملف إدخال يحتوي كل سطر من أسطره على اسم لا يزيد طوله عن 20 حرفاً. أجر التعديل اللازم على برنامج إنشاء الهيكل الشجري الثنائي في السؤال السابق بحيث تقرأ هذه الأسماء وتدخل إلى الهيكل الشجري الثنائي، بحيث تطبع الأسماء مرتبة ترتيباً تصاعدياً عند تطبيق طريقة الاستعراض وفق السياق الوسطي.



أسئلة التقويم الذاتي (2)

1. ما ترتيب استعراض عناصر الهيكل الشجري الثنائي التالي عند تطبيق طريقة الاستعراض وفق السياق الوسطي (Inorder Traversal):



2. ما ترتيب استعراض عناصر الهيكل الشجري الثنائي الوارد في سؤال التقويم الذاتي (1) أعلاه عند تطبيق طريقة الاستعراض وفق السياق التبعي؟
3. ما ترتيب استعراض عناصر الهيكل الشجري الثنائي الوارد في سؤال التقويم الذاتي (2) أعلاه عند تطبيق طريقة الاستعراض وفق السياق القبلي؟
4. افرض أن التسلسل لعناصر هيكل شجري ثنائي عند استعراض عناصره وفق السياقين القبلي والوسطي هو كما يلي: التسلسل وفق السياق القبلي:

G, B, Q, A, C, K, F, P, D, E, R, H

التسلسل وفق السياق الوسطي:

Q, B, K, C, F, A, G, P, E, D, H, R

ارسم هذا الهيكل الشجري.

5. ارسم الهيكل الشجري الثنائي الذي يخزن التعبير الحسابي التالي:

$$(A - (B - C)) * (D + E)$$

6. ما الهيكل الشجري الثنائي الناتج عند تغير الأقواس في السؤال السابق بحيث يصبح التعبير الحسابي كما يلي:

$$A - (B - C) * D + E$$

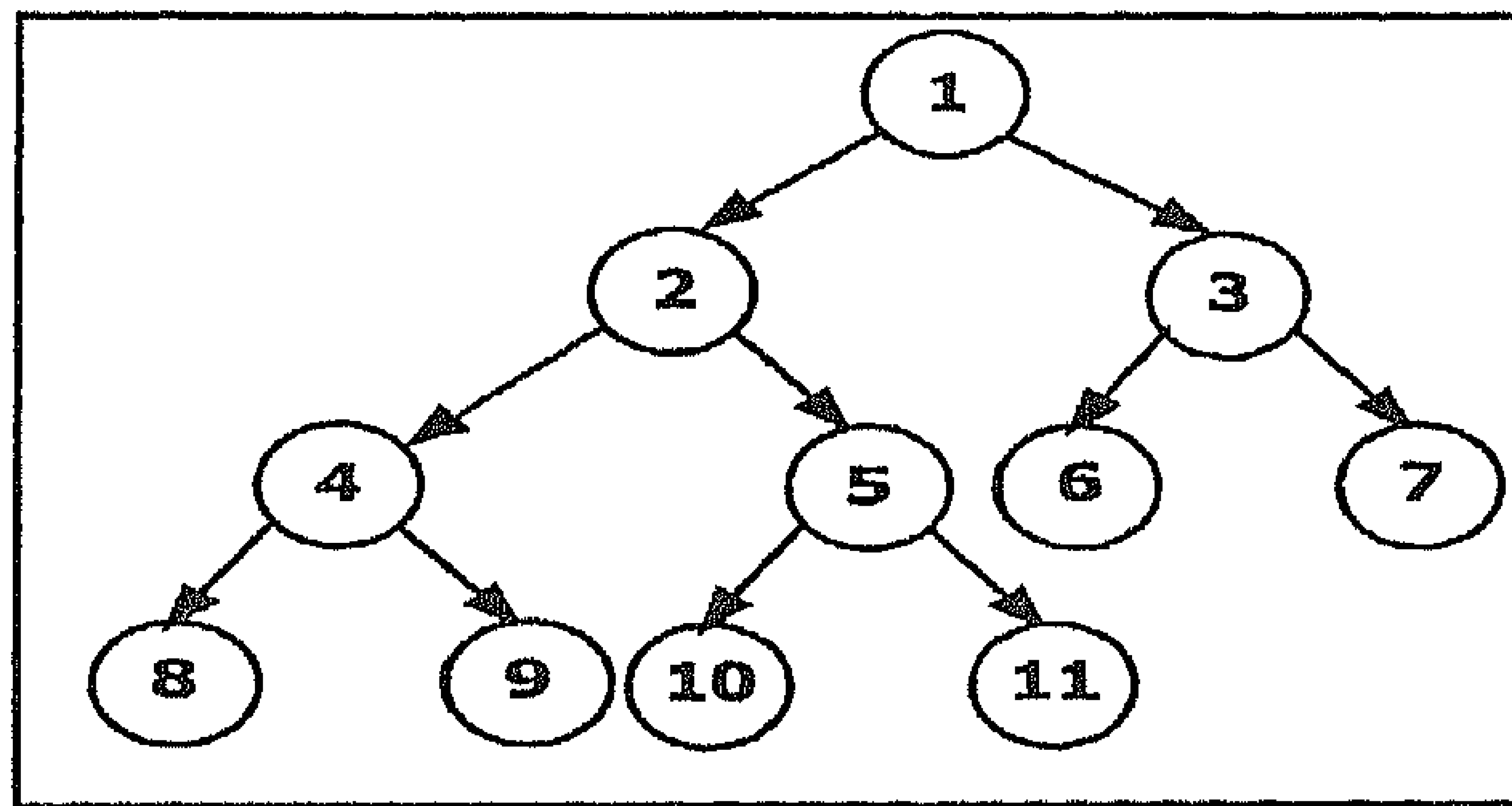
5. أنواع الهياكل الشجرية الثنائية

1.5 الهياكل الشجرية الثنائية الكاملة
(Complete Binary Tree)

كما أسلفنا، يمكن للعنصر الواحد في الهيكل الشجري الثنائي أن يحتوي على ابنين اثنين على الأكثر. وبناءً على ذلك فإنه من الواضح أن أكبر عدد من الأبناء يمكن أن يتوفر في أي مستوى (i) هو (2^{i-1})على فرض أن الجذر يقع على المستوى 1.

يقال للهياكل الشجرية الثنائية أنها كاملة إذا كانت جميع المستويات (مع احتمال استثناء المستوى الأخير) تحتوي على أكبر عدد ممكن من العناصر (أي أن كل مستوى (i) من هذه المستويات يحتوي على عدد من العناصر يساوي (2^{i-1}). وكذلك فإن العناصر على المستوى الأخير تظهر من أقصى اليسار إلى اليمين.

فعلى سبيل المثال لو فرضنا أن هناك هيكلاً شجرياً ثنائياً كاملاً يحتوي على 11 عنصراً يمكن الإشارة إليه بالرمز T11 ويمكن تمثيله على النحو التالي: الشكل (31)



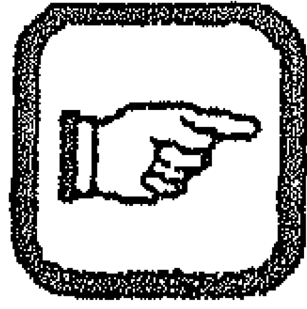
الشكل (31): هيكل شجري ثنائي كامل (T11)

ولهذا فإن هناك هيكل شجري ثنائي كامل فريد من نوعه ومميز عن غيره يحتوي على عدد (n) من العناصر. بمعنى آخر، لا يوجد أكثر من هيكل شجري كامل واحد يحتوي على عدد (n) من العناصر.

وفي الشكل الثنائي T11، قمنا بترقيم العناصر بالأعداد من 1 إلى 11 ابتداءً من الجذر ومن مستوى إلى المستوى الذي يليه، ومن اليسار إلى اليمين، حيث أنه من هذا الترقيم يمكن معرفة أبناء ووالد أي عنصر من العناصر.

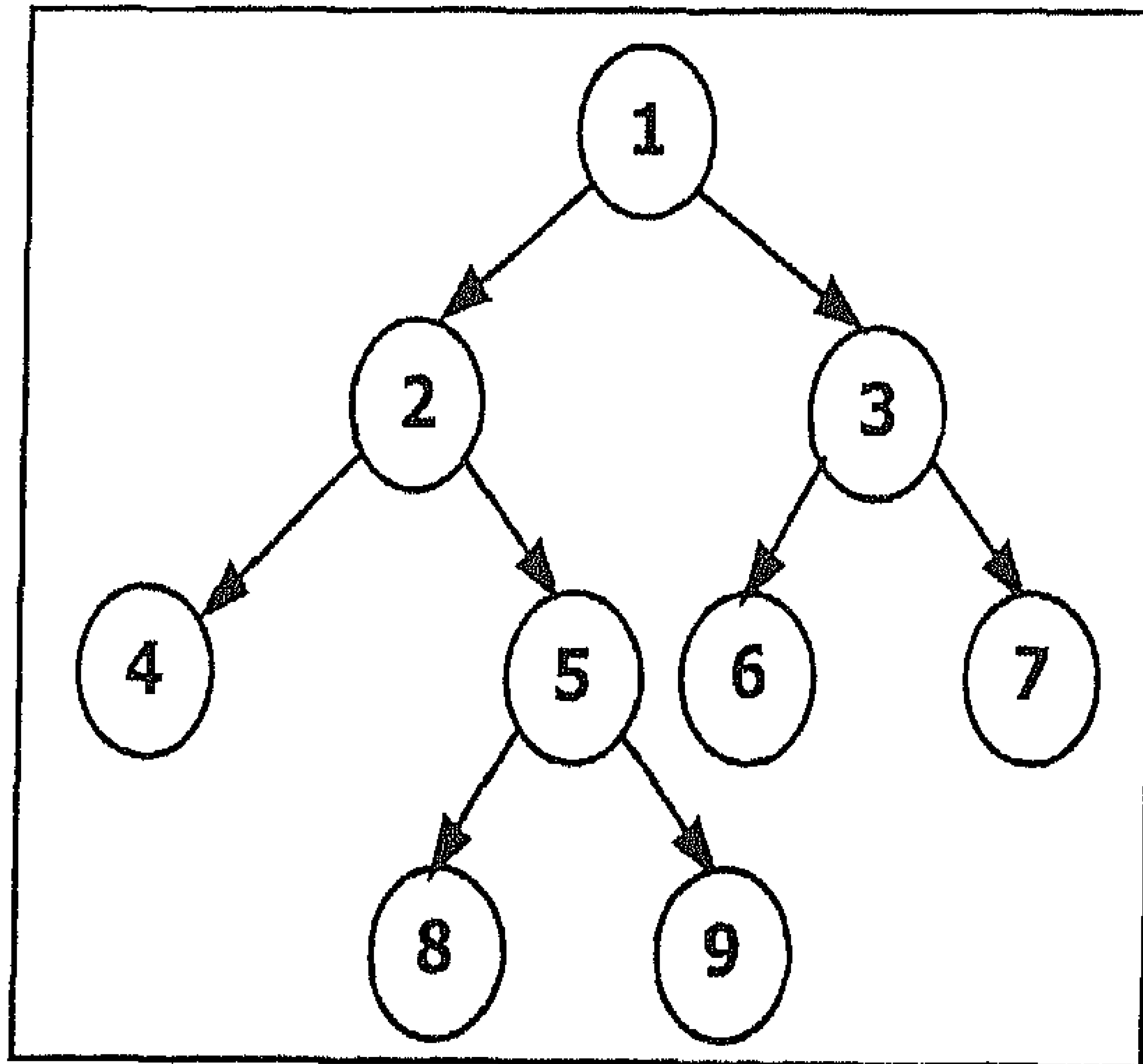
وكما ذكرنا سابقاً،عزيزي الدارس، فإن الابن الأيسر للعنصر (i) يقع في الموقع (2*i)، والابن الأيمن للعنصر (i) يقع في الموقع (2*i+1) . وأما والد العنصر (i) فهو يقع في الموقع [i / 2]، بحيث يعني الرمز [i / 2] أكبر عدد صحيح يساوي أو أقل من (i / 2).

فالابن الأيمن للعنصر الذي يشار إليه بالرقم 5 على سبيل المثال يحمل الرقم 5*2 أي 10 والابن الأيمن للعنصر نفسه يشار إليه بالرقم 11. أما والد هذا العنصر فيشار إليه بالموقع [5/2] ويساوي (2).



مثال (14)

الهيكل الشجري التالي (الشكل (32)) غير كامل وذلك لكون العناصر في المستوى الأخير لا تسير من اليسار إلى اليمين دون ترك فراغات.



الشكل (32)

وتكمن أهمية الأشجار الثنائية الكاملة في كونها أفضل لعملية البحث حيث يتطلب الإجراء (Search & ST) عدداً من المقارنات مساوياً لعمق الشجرة أي (O (Log2N حيث (N) عدد العناصر في الهيكل الشجري. مما يعني أن الوقت اللازم لتنفيذ (Search & ST) على هيكل شجري كامل هو (log2N). اما في أسوء الأحوال فإن عمق الشجرة الثنائية هو (N) كما في الشجرة المبينة في الشكل (24) وعليه فإن (Search & ST) قد يتطلب (N) من المقارنات للوصول إلى العنصر المطلوب وبهذا يكون Search & ST من الدرجة O(N) في أسوء الأحوال. ولكون الدالتان insertNode وdeleteNo يقومان بالبحث أولا عن مكان العنصر المراد إضافته أو حذفه ثم تبقى عمليات بسيطة لتفسر قيم المؤشرات فإن الوقت اللازم لتنفيذ الدالتين هو O(log2N) إذا كانت الشجرة كاملة وتلك هي الحالة الأفضل أما إذا لم تكن كاملة فإن الدالتين قد يتطلبا وقتاً متناسباً مع O(N) في أسوء الأحوال وذلك إذا كانت الشجرة كما في الشكل (24).

2.5 الهياكل الشجرية الثنائية العشوائية

يمكن أن نسأل السؤال التالي: هل من المفيد أن نحافظ على الهيكل الشجري متوازناً عند الإضافة إليه أو الحذف منه أم أن ذلك غير مفيد؟

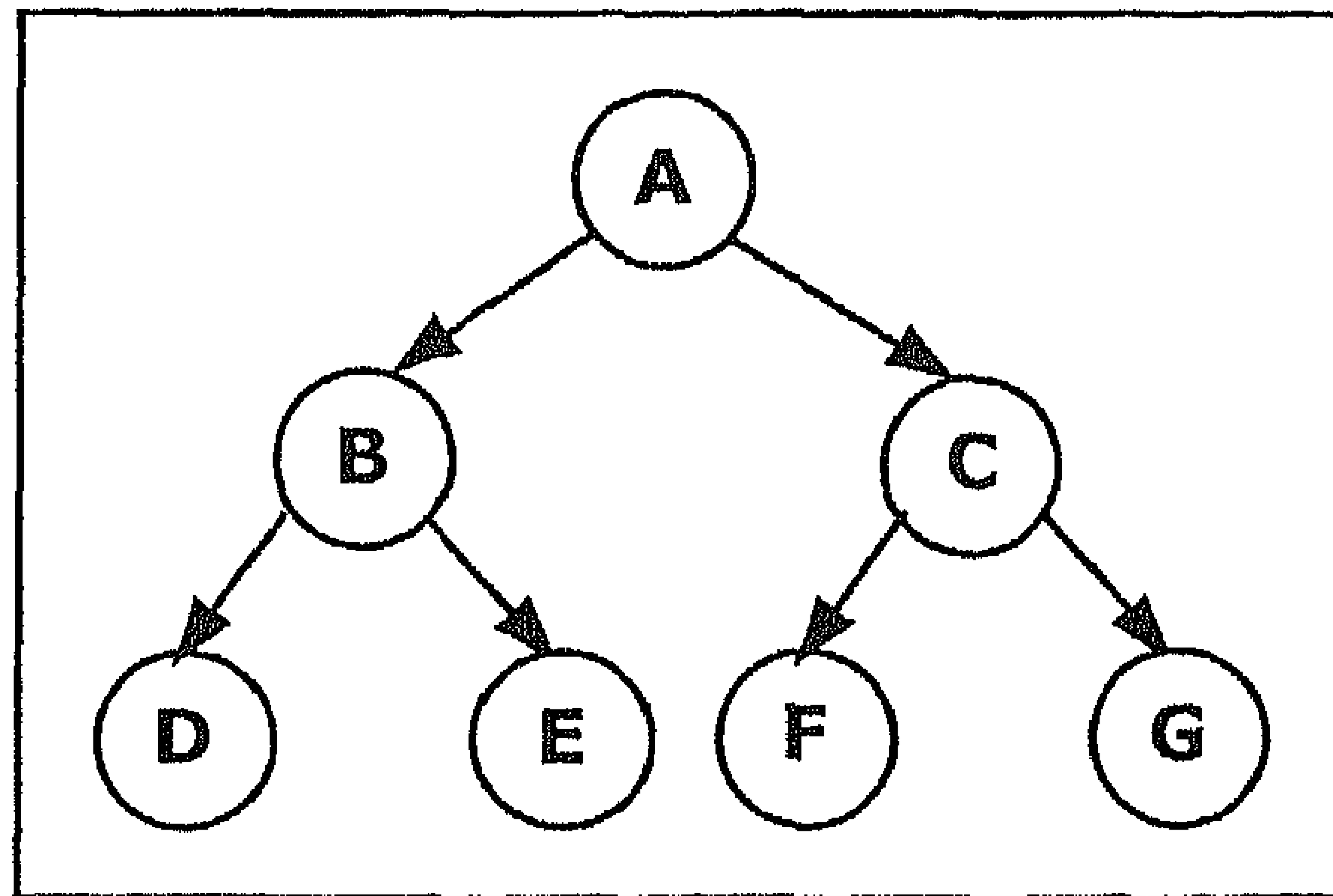
كما تعلم، عزيزي الدارس، فإنه عند الإضافة إلى الهيكل الشجري ربما تكون القيم المراد تخزينها في عناصر الهيكل الشجري عشوائية الترتيب مما ينتج هيكلاً شجرياً غير متوازن. فإذا كان الجواب على السؤال السابق بالإيجاب، فهذا يتطلب منّا إعادة موازنة الهيكل الشجري ربّما بعد كل عملية إضافة أو عملية حذف من الهيكل الشجري.

وقد أثبتت الدراسات بأن عدد عمليات المقارنة في حالة الاستقصاء والبحث عن عنصر معين في هيكل شجري ثنائي عشوائي تزيد بمعدّل يقارب 40 % عن عدد عمليات المقارنة في حالة كون الهيكل الشجري متوازناً.

فإذا كان التطبيق الذي نحن بصدده يتطلب كفاءةً عالية في الإنجاز من حيث الوقت، يجب مقارنة تكلفة موازنة الهيكل الشجري مع أهمية هذا التطبيق، حيث أن موازنة الهيكل الشجري يتطلب جهداً كبيراً في إعداد البرامج اللازمة لذلك.

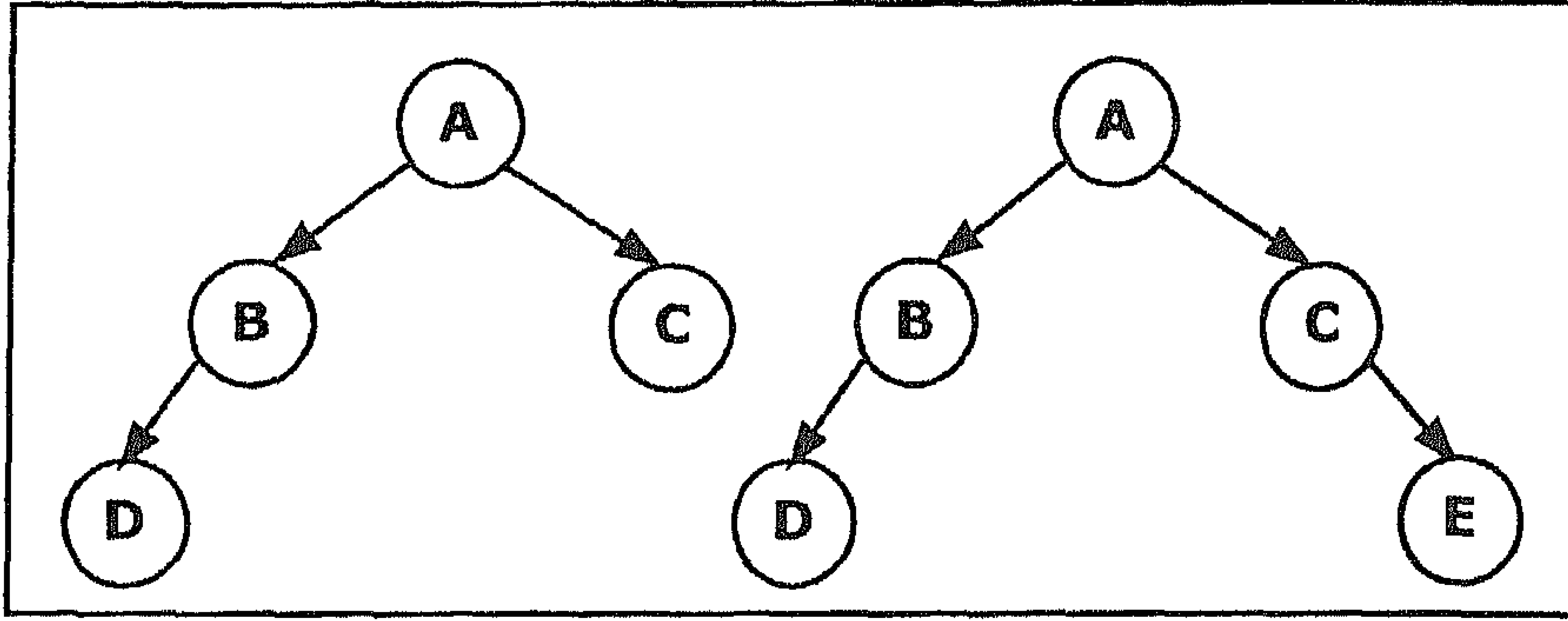
3.5 الهياكل الشجرية الثنائية المتوازنة (Balanced Binary Trees)

يقال عن الهيكل الشجري الثنائي أنه متوازن، إذا انبثق من كل عنصر من العناصر عنصران اثنان أو لم ينبثق منها أي عنصر على الإطلاق، وكذلك فإن جميع العناصر الورقية تقع على المستوى نفسه. فعلى سبيل المثال فإن الهيكل الشجري التالي يعتبر متوازناً. ويطلق على مثل هذا النوع من الهياكل الشجرية المتوزانة بالهياكل الشجرية المتوازنة بالكامل (Completely Balanced) أو (Fully Balanced) والشكل (33).



الشكل (33)

بينما الهياكل الشجرية التالية (الشكل 34) غير متوازنة بالكامل وذلك لعدم انطباق التعريف السابق عليها.



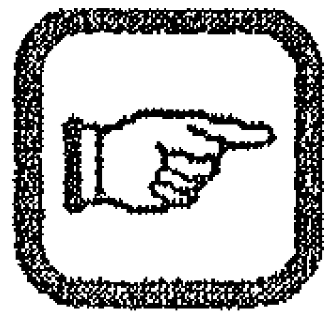
الشكل (34)

ويوجد نوع آخر من الهياكل الشجرية المتوازنة يطلق عليها اسم الهياكل الشجرية المتوازنة من حيث الارتفاع أو العمق (Height-Balanced Trees)، وتعرف بالاسم AVL-Tree والاختصارات AVL أخذت من اسم العالمين الرياضيين الروسيين Adel'son-Vel'Slkii وLandis، حيث تم وصف مثل هذا الهيكل الشجري عام 1962. والهدف الرئيس من الهياكل الشجرية المتوازنة هو إنجاز عمليات الاستقصاء والإضافة والحذف في أقل وقت ممكن. فلو فرضنا أن عدد العناصر في الهيكل الشجري هو (n)، فيمكن إنجاز مثل هذه العمليات في وقت لوغاريتمي أي (O(Log n، وحتى في أسوء الأحوال (Worst Case).

ومع أن الهيكل الشجري الثنائي المتوازن بالكامل يعتبر مثالياً من حيث كفاءة الوقت عند إنجاز العمليات الآنفة الذكر، إلاّ أنه ليس من السهل الحصول على مثل هذا الهيكل الشجري والذي يتميز بأن الشجيرة اليسرى والشجيرة اليمنى لأي عنصر لهما العمق أو الارتفاع نفسه، بل من الممكن ومن السهل الحصول على هيكل شجري آخر يطلق عليه شجيرة (AVL)، والميزة فيه هي أن ارتفاع الشجيرة اليمنى لأي عنصر من عناصرها لا يمكن أن يختلف بأكثر من عنصر واحد عن ارتفاع الشجيرة اليسرى بالزيادة أو النقصان.

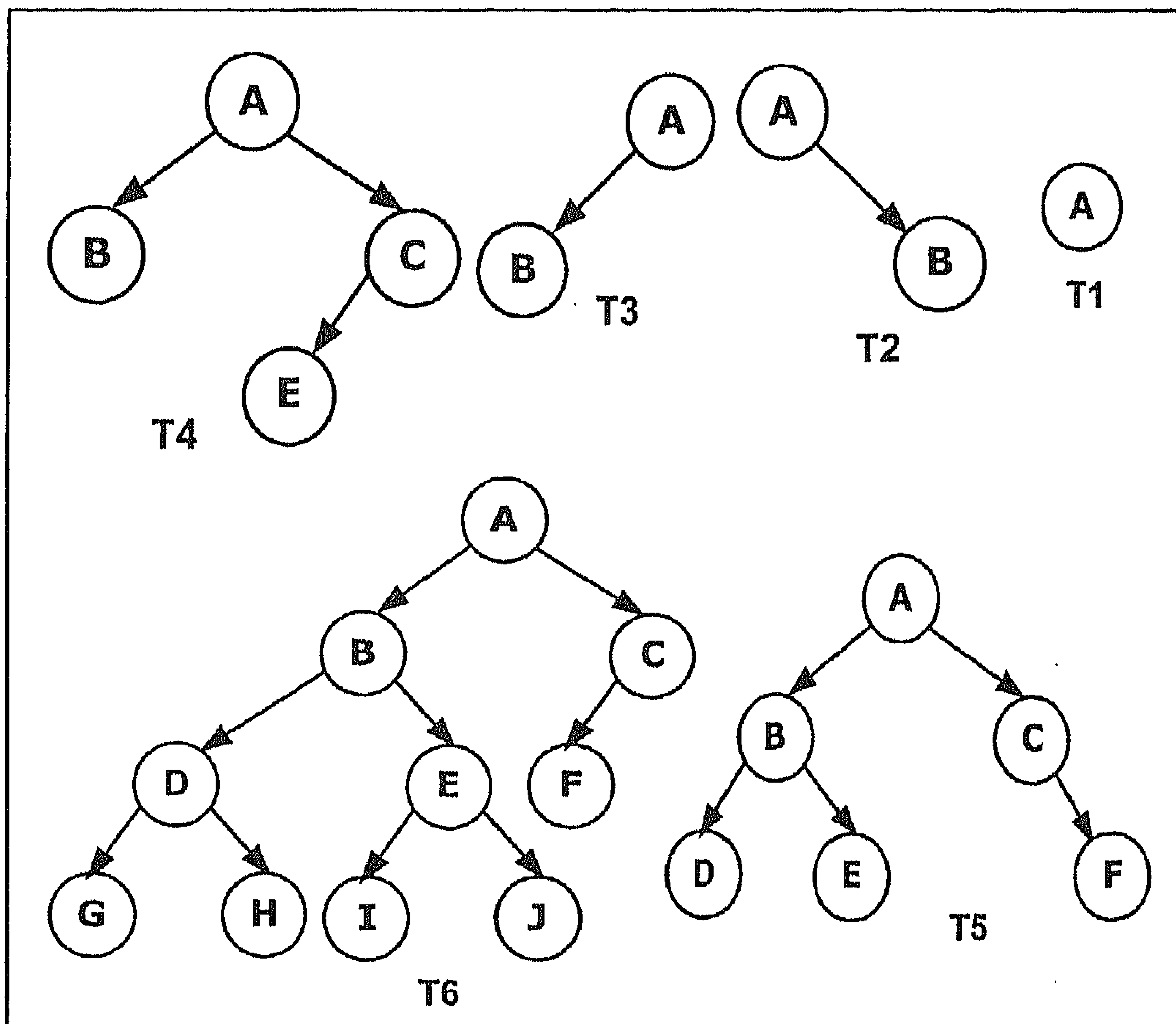
ومن هذا المنطلق يمكننا، عزيزي الدارس، تعريف شجيرة (AVL) أو الشجيرة المتوازنة من حيث العمق (Height-Balanced Binary Tree) كما يلي:

إذا كان الهيكل الشجري الثنائي خالياً من العناصر، فيعتبر متوازناً من حيث العمق. أما إذا كان الهيكل الشجري غير خالٍ فيعتبر متوازناً من حيث العمق إذا كان عمق الشجيرة اليمنى وعمق الشجرة اليسرى للجذر لا يختلفان بأكثر من عنصر واحد على الأكثر. وكذلك فإن الشجيرة اليمنى والشجيرة اليسرى تعتبران شجيرتا ((AVL. ويرتبط مع كل عنصر من عناصر شجيرة (AVL) معامل توازن (Balance Factor) حيث أنه إذا كان عمق الشجيرة اليمنى للعنصر يزيد عن عمق الشجيرة اليسرى بمقدار 1 فيقال عن معامل التوازن أنه عميق لليمين (Right High)، أما إذا كان عمق الشجيرة اليسرى يزيد بمقدار 1 عن عمق الشجيرة اليمنى فيقال عن معامل التوازن بأنه عميق لليسار (Left High)، وأما إذا كان عمق الشجيرتين متساوٍ فيقال عن معامل التوازن بأنه متساوٍ.

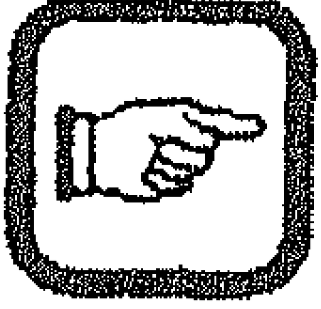


مثال (15)

تعتبر الهياكل الشجرية التالية من نوع (AVL) (الشكل (35)):



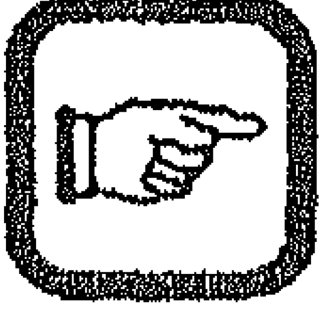
الشكل (35)



مثال (16)

تعتبر الهياكل الشجرية التالية من نوع ((AVL المنحرفة إلى اليسار، (الشكل (36))

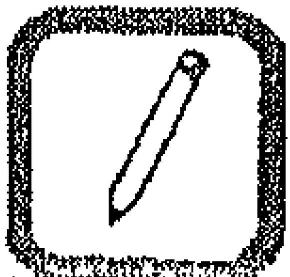
الشكل (36)



مثال (17)

لا تعتبر الهياكل الشجرية التالية من نوع (AVL) (الشكل (37)).

الشكل (37)



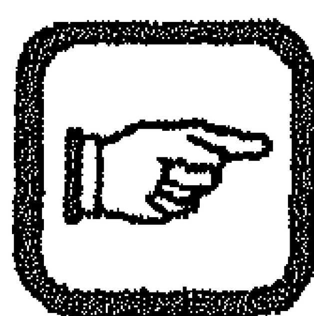
تدريب (15)

ارسم هيكلاً شجرياً متوازن العمق لأشهر السنة.

الهياكل الشجرية الثنائية المقيدة (HEAP)

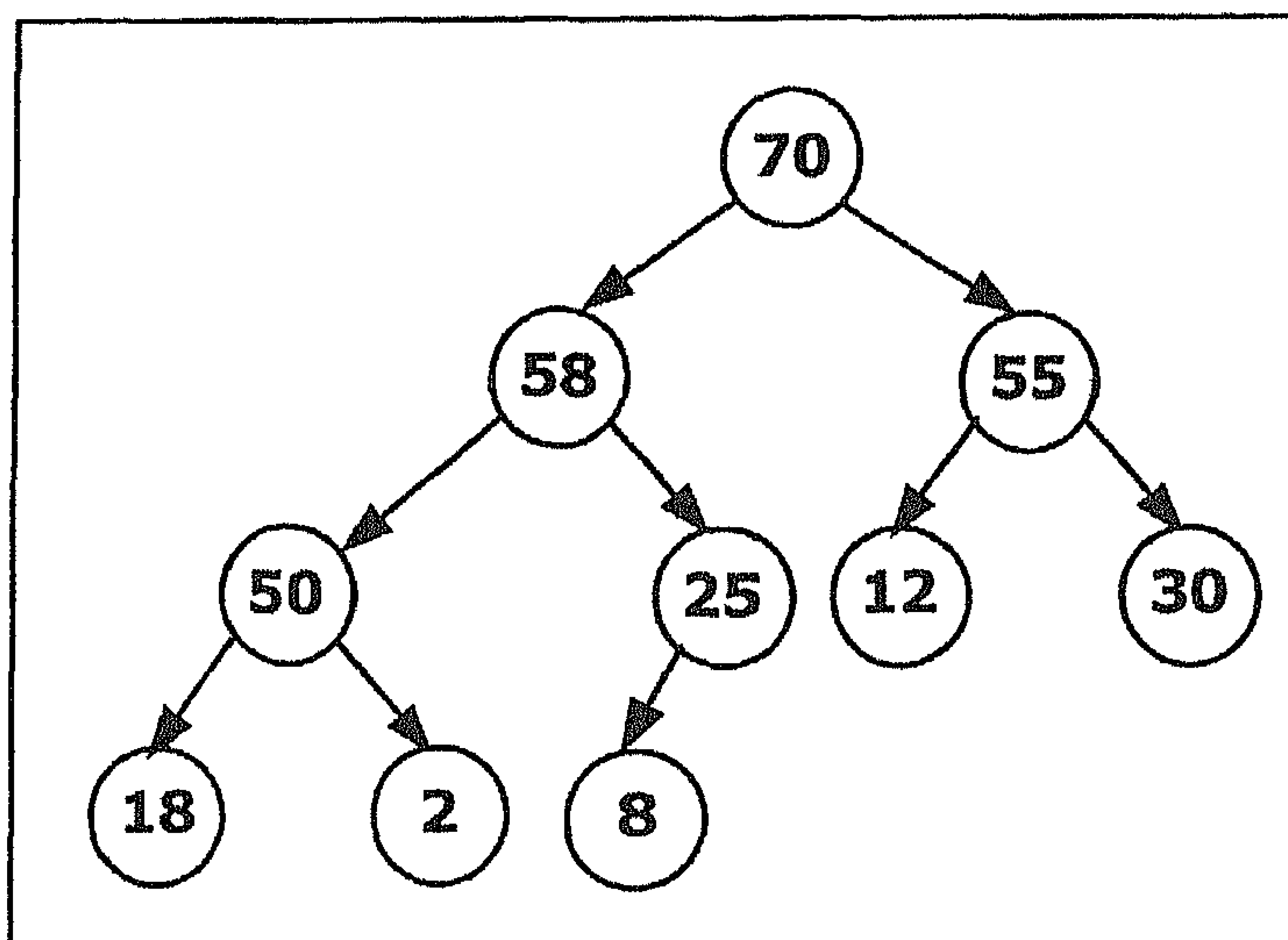
يستعمل هذا النوع من الهياكل الشجرية الثنائية في الفرز المقيد (Heap Sort) ويمكن تعريف الهيكل الشجري الثنائي المقيد بأنه هيكل شجري ثنائي كامل (Compelete Binary Tree) يتميز بالخاصية التالية: القيمة المخزنة في أي عنصر من العناصر يجب أن تكون مساوية أو أكبر من القيم المخزنة في العنصرين اللذين يمثلان ابني العنصر إذا كان لهذا العنصر أبناء.

وهذا يعني بأن القيمة المخزنة في الجذر تمثل أكبر القيم المخزنة في عناصر الهيكل الشجري الثنائي.



مثال (18)

يعتبر الهيكل الشجري الثنائي التالي من النوع المقيد، (الشكل (38)):



الشكل (38)

وكما تعلم فإنه نتيجة لتنفيذ عملية الفرز المقيد، يتم طباعة القيم الموجودة في العناصر مرتبةً ترتيباً تنازلياً.



أسئلة التقويم الذاتي (3)

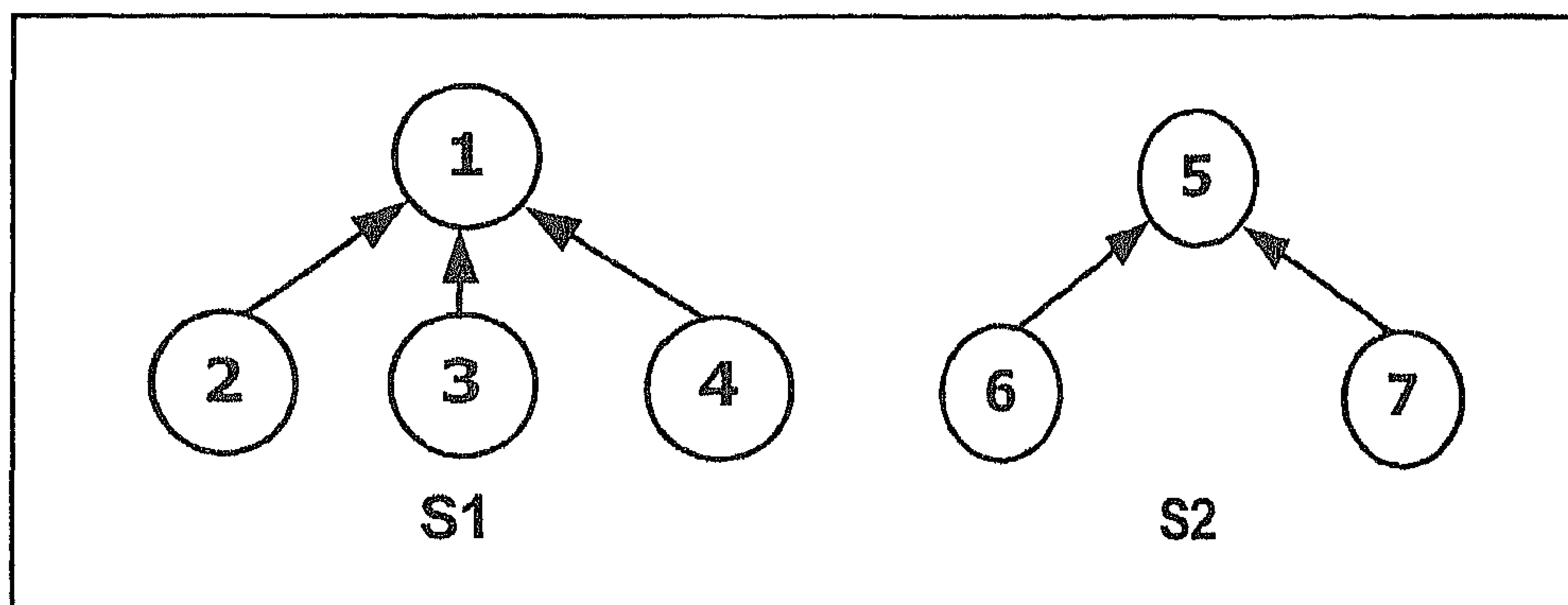
اذكر أربعة أنواع من الهياكل الشجرية الثنائية مع ذكر خصائص كل نوع من هذه الأنواع. يمكنك الاستعانة بالرسم.

6. تطبيقات على الهياكل الشجرية

هناك، عزيزي الدارس، تطبيقات عدة تستعمل فيها الهياكل الشجرية نذكر منها ما يلي:

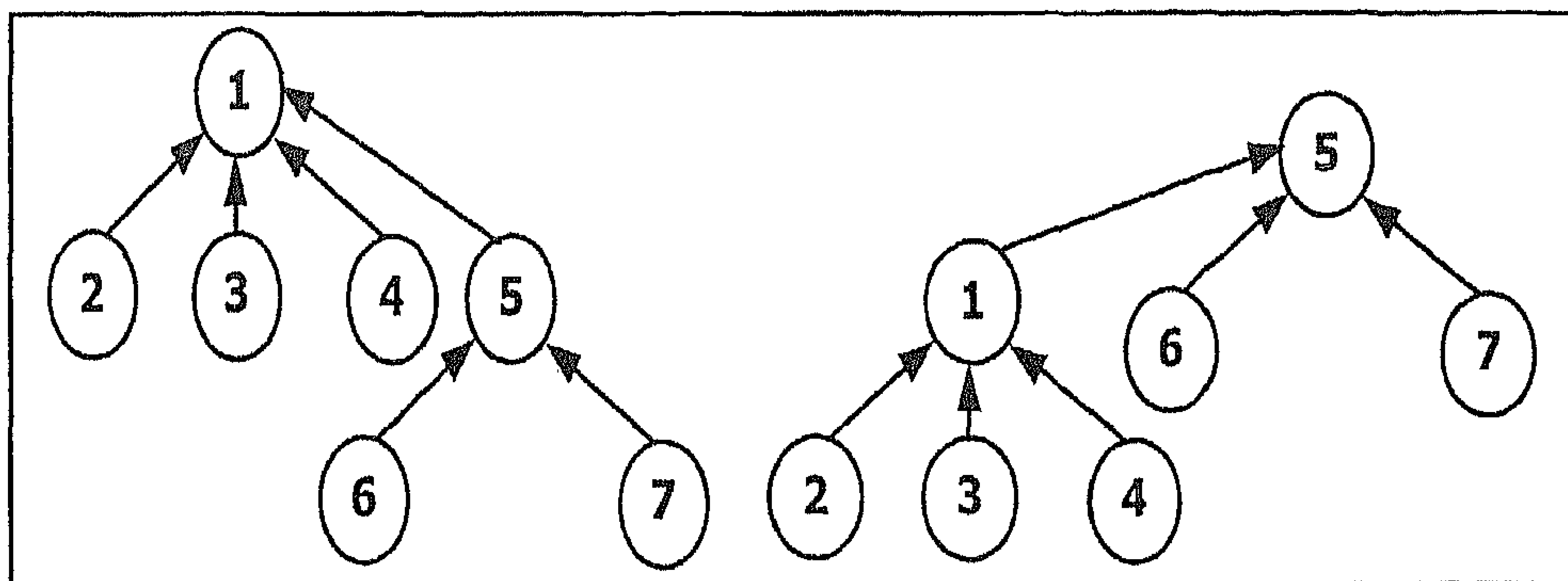
1.6 تمثيل المجموعات (Set Representation)

في هذه الحالة يمكن ربط العناصر مع بعضها البعض بحيث أن عناصر المجموعة الواحدة يمكن ربطها إلى عنصر الوالد على العكس من الشيء الدارج. لو فرضنا أن لدينا المجموعة (S1) وعناصرها {1,2,3,4} والمجموعة (S2) وعناصرها {5,6,7}. يمكن تمثيل المجموعة (S1) والمجموعة (S2) كما يلي، الشكل (39)):



الشكل (39)

وباستخدام هذا التمثيل، يمكن التعبير عن اتحاد S1 وS2 (S1 U S2) بالشكل التالي، (الشكل(40)):

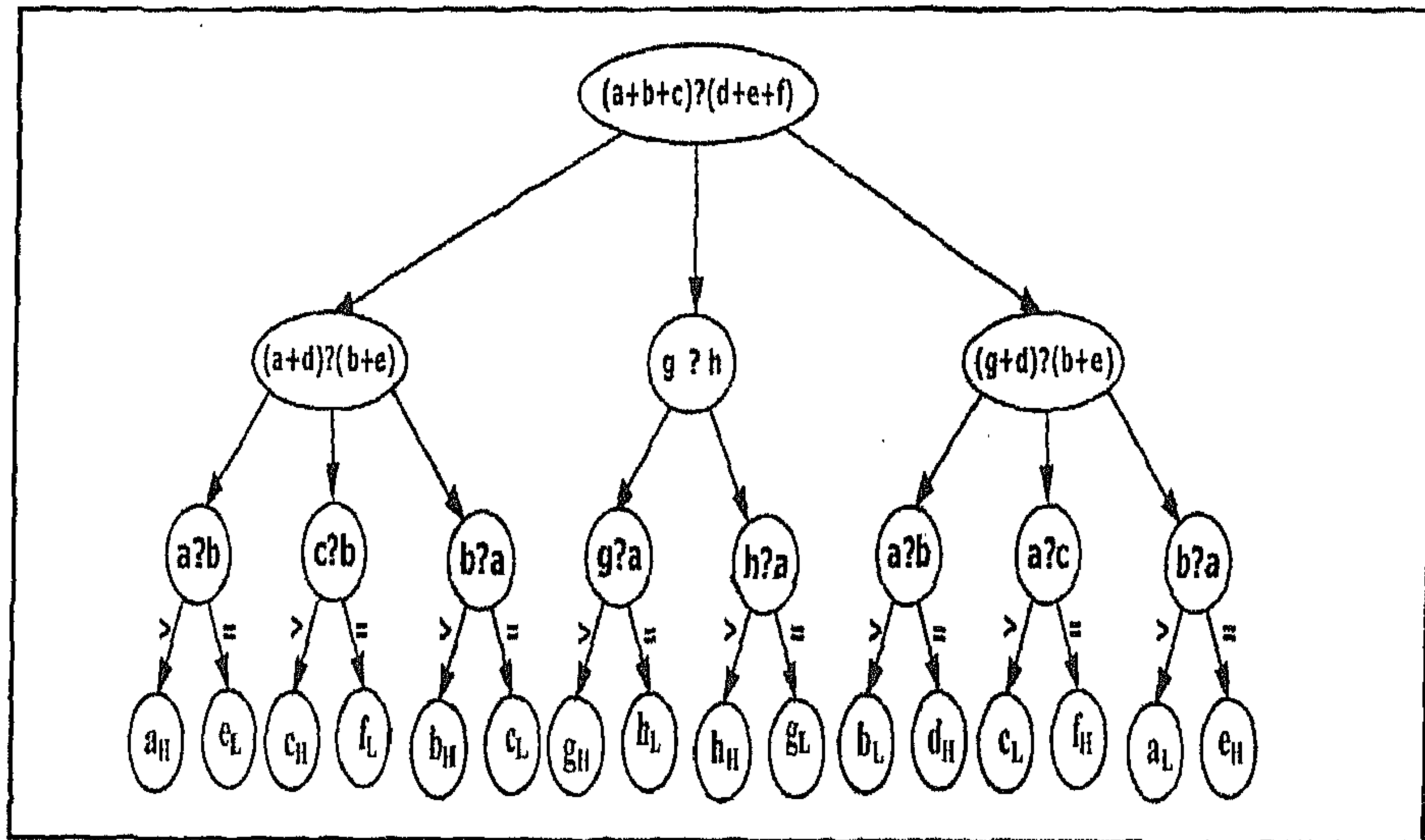


أو

الشكل (40)

2.6 استخدام الهياكل الشجرية للمساعدة في اتخاذ القرار

ومن الأمثلة الشائعة على ذلك مسألة العملات الثماني (Coins Problem8) وتتلخص هذه المسألة بوجود ثماني عملات (a,b,c,d,e,f,g,h)، وبوجود عملة واحدة من بينها مزورة يختلف وزنها عن وزن العملات الأخرى، ونرغب في تحديد العملة المزيفة واستخراجها من بين العملات الأخرى بأقل عدد من المقارنات، وفي الوقت نفسه معرفة ما إذا كانت العملة المزيفة أثقل أو أخف من العملات المتبقية. يمكن القيام بهذا العمل عن طريق الهيكل الشجري التالي، (الشكل (41)) والذي يحدد أيضاً ما إذا كانت العملة المزيفة أثقل أم أخف من العملات الأخرى:



الشكل (41)

حيث (ah) تعني بأنه إذا تم الوصول إلى هذا المكان فإن العملة (a) هي المزيفة وهي أثقل من العملات الأخرى، بينما (aL) تعني أن a هي العملة المزيفة وأنها أخف من العملات الأخرى وهكذا.

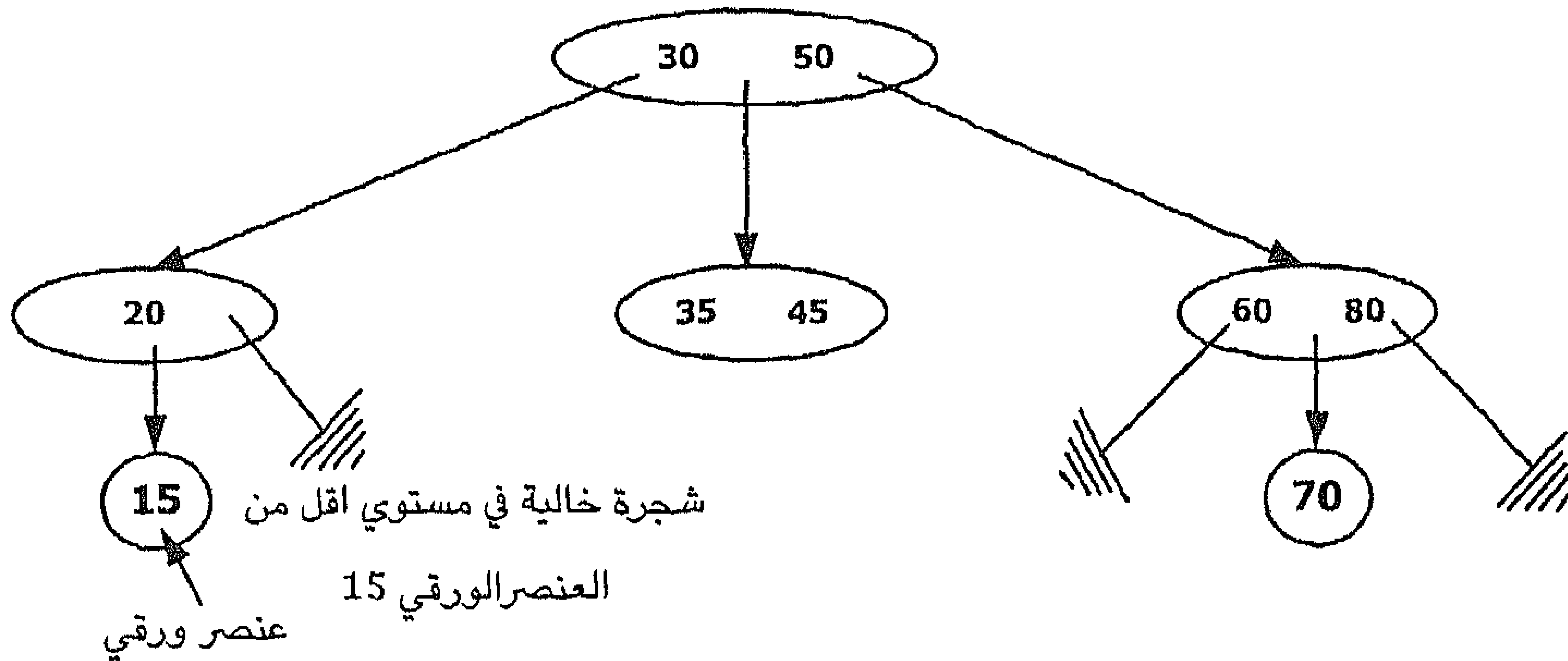
3.6 ألعاب متفرقة

ومن الاستخدامات المهمة للهياكل الشجرية في شجيرات الألعاب مثل ألعاب الشطرنج (Chess)، ولعبة تيك-تاك-تو (tic-tac-toe)، ولعبة Checkers, go, Kalah (Nim)، وما إلى ذلك.

7. الهياكل الشجرية نوع B-Tree

تعتبر الهياكل الشجرية من نوع B-tree من أهم أنواع الهياكل الشجرية بالإضافة إلى الهياكل الشجرية الثنائية، حيث أنها تستخدم في تخزين بعض المعلومات المهمة عند التعامل مع الملفات وقواعد البيانات، وخاصةً عند تمثيل الفهارس. ونظراً لأهمية هذا النوع من الهياكل الشجرية في التطبيقات العملية سنحاول،عزيزي الدارس، التطرق إليها بشيء من التفصيل.

يقال للهيكل الشجري بأن رتبته d(order or degree) إذا كان عدد الأبناء لأي عنصر من عناصر هذا الهيكل الشجري لا يزيد على (d)، حيث تمثل (d) عدداً صحيحاً. فلو فرضنا أن (k) يمثل عدد الأبناء لعنصر ما (بحيث أن k <= d) فإن هذا العنصر يحتوي على (k-1) من القيم المسماة أدلة (Keys) والتي تستخدم عند إجراء عملية الاستقصاء على الهيكل الشجري. وتستخدم هذه الأدلة الموجودة في العنصر لتقسيم العناصر المنبثقة من هذا العنصر إلى (k) من المجموعات الجزئية كما هو مبين في الشكل (42) التالي:



الشكل (42): شجيرة استقصاء بثلاث طرق (A 3-way Search Tree)

ومن خواص هذا النوع من الهياكل الشجرية، أن جميع العناصر الأقل من قيمة أي من الأدلة ترتبط بالمؤشر إلى يسار تلك القيمة. أما القيم الأكبر من قيمة ذلك الدليل والأصغر من القيمة التالية من قيم الأدلة في ذلك الموقع، فترتبط بالمؤشر الواقع إلى يمين الدليل الأقل من تلك القيم وإلى يسار الدليل الأكبر من تلك الأدلة. فعلى سبيل المثال، وبالرجوع إلى الشكل (42) نجد أن الجذر يحتوي على الدليلين 30 و50 وأن المؤشر الواقع إلى يسار القيمة 30 وإلى يسار الدليل 50 فيربط الأدلة 35 و45 وهذه الأدلة أكبر من القيمة 30 وأصغر من القيمة 50.

أما بالنسبة لرتبة هذا الهيكل الشجري، فتساوي 3، حيث نجد أن بعض الأبناء تمثل بالقيمة NULL ونلاحظ أيضاً أن رتبة الهيكل الشجري مساوية لأكبر عدد من الأدلة الموجودة فـي أي عنصر من العناصر مضافاً إليها 1 وبما أنه يوجد اثنين من الأدلة على الأكثر من أي من العناصر، لذلك فإن رتبة هذا الهيكل الشجري تساوي 1+2 ويساوي 3 ويقال عن هذا الـنوع من الهياكل الشجرية بأنه ثلاثي الاتجاه عند الاستقصاء (3-Way Search Tree).

والهدف الذي نسعى إليه من هذه الهياكل الشجريّة هو تخزين مثل هذه الأدلة في هيكل شجري استقصائي متعدد الاتجاهات بحيث نقلل عدد عمليات الوصول إلى الملفات، لأن عملية الوصول إلى الملف لإحضار مثل هذه القيم تستغرق الكثير من الوقت. وبما أن عملية الوصول تتم في العادة حسب المستوى، أي المستوى الأول أولاً ثم المستوى الثاني وهكذا، فلا شك أن هدفنا هو تقليل عمق أو ارتفاع الهيكل الشجري. ويمكن تحقيق ذلك كما يلي:

أولاً: عدم وجود شجيرات خالية على مستوى أقل من مستوى العناصر الورقية وينشأ عن هذا تقسيم قيم الأدلة بأعلى كفاءة ممكنة. فعلى سبيل المثال نجد أن التقسيم في الشكل السابق لا يعطي أقصى كفاءةٍ ممكنة، حيث أن الشجيـرة اليمنى للعنصر (20) خالية والشجيرة اليسرى غير خالية. (انظر الشكل (42)) ففي هذا الشكل إذا استطعنا وضع العنصر 15 إلى جانب العنصر 20 تصبح الشجيرة أكثر كفاءة من حيث التخزين وسرعة البحث.

ثانياً: جعل العناصر الورقية على المستوى نفسه بحيث أن الوصول إليها يحتاج إلى الوقت نفسه.

ثالثاً: كل عنصر من العناصر باستثناء العناصر الورقية يحتوي على عددٍ من قيم الأبناء لا يقل عن نصف أكبر عدد ممكن للأبناء في أي من العناصر.

هذه المتطلبات وبعض الصفات الأخرى نوجزها،عزيزي الدارس، بما يلي وهي بمجموعها تعطي تعريف الهياكل الشجرية نوع-B التالي:

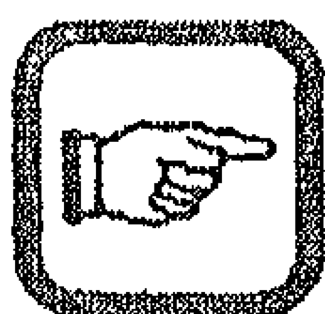
يعتبر الهيكل الشجري من نوع-B وبرتبة d إذا حقق الشروط التالية:

1. يحتوي كل عنصر من العناصر باستثناء العناصر الورقية على عدد من الأبناء يزيد بمقدار 1 عن عدد الأدلة (Keys) حيث تقسم هذه الأدلة أدلة الأبناء بحيث تسهل عملية الاستقصاء بأقل وقت ممكن.

2. تقع جميع العناصر الورقية على المستوى نفسه.

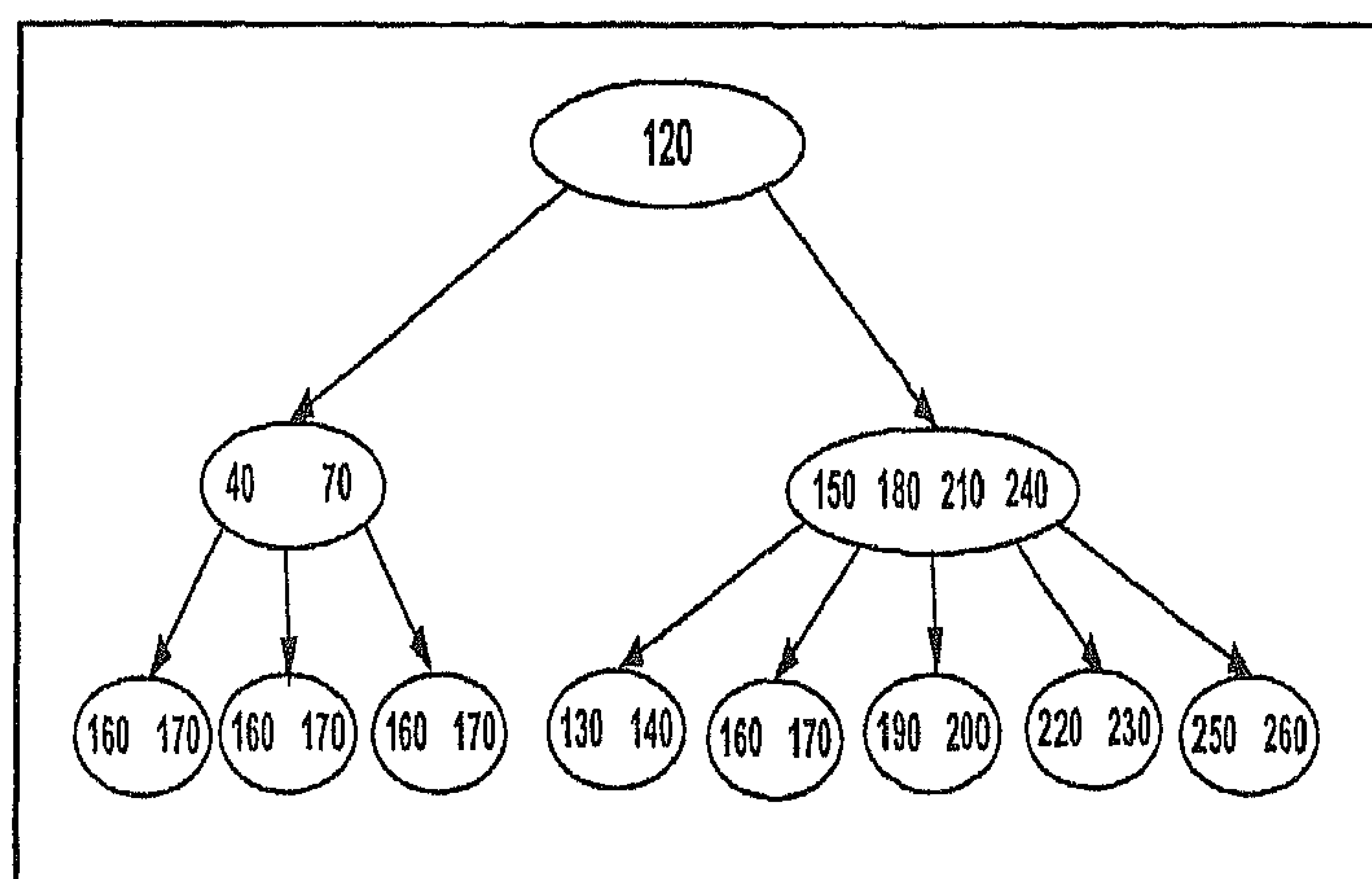
3. باستثناء الجذر ينبثق من العناصر الأخرى عدد من الأبناء لا يزيد عن رتبة الهيكل الشجري d ولا يقل عن [d/2].

4. يمكن أن ينبثق عن الجذر عدد d من الأبناء، ويمكن لعدد هذه الأبناء، أن يصل إلى 2 إذا لم يكن الجذر هو العنصر الوحيد.



مثال (19)

الهيكل الشجري التالي من نوع-B ويحمل الرتبة 5:

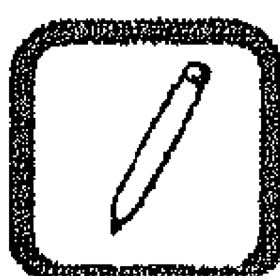


الشكل (43)

كما أشرنا، عزيزي الدارس، يعتبر الهيكل الشجري من نوع-B هيكلاً شجرياً استقصائياً. فإذا أخذنا بعين الاعتبار الجذر ويحتوي على الدليل (Key120)، فإن جميع الأدلة الواقعة في الشجيرة اليسرى للجذر أقل من 120، وجميع الأدلة الواقعة في الشجيرة اليمنى أكبر من 120. وهذه الخاصية تنطبق على كل عنصر من العناصر باستثناء العناصر الورقية حيث لا ينبثق منها أبناء.

ولو أخذنا العنصر الذي يحتوي على القيم (150 180 210 240) فسوف نجد أن مؤشر الربط إلى يسار القيمة 150 يشير إلى الأدلة التي قيمتها أصغر من 150، وأن المؤشر إلى يمين القيمة 150 يوصل إلى القيم التي تزيد عن 150، وبما أن المؤشر إلى يمين القيمة 150 هو أيضاً إلى يسار الدليل 180، فهذا يعني أن مؤشر الربط هذا يوصل إلى الأدلة التي تزيد عن 150 وتقل عن 180. وبالأسلوب نفسه يمكن البحث والاستقصاء عن أي قيمة من الأدلة في وقت قصير.

وعند الوصول إلى العنصر الذي يحتوي على الدليل المطلوب يتم مقارنة القيمة المعطاة مع القيم الموجودة في العنصر وتحديد هذه القيمة.



تدريب (16)

ارسم هيكلاً شجرياً من النوع-B رتبته 5 وعمقه 3 بحيث يحتوي على الأحرف الأبجدية الإنجليزية.

8. الخلاصة

لقد درسنا في هذه الوحدة موضوعاً مهماً، في تركيب البيانات، وبخاصة الهرمية منها هو: الهياكل الشجرية. وأوصيك الآن،عزيزي الدارس، بمراجعة الأهداف التعليمية المشار إليها في بند 2.1. إذا كان بمقدورك تحقيق هذه الأهداف تكون قد استوعبت موضوع هذه الوحدة. وإلا فلا بأس عليك، أعد دراسة النص العلمي بما في ذلك الأمثلة والتدريبات. بعد ذلك يمكنك مقارنة حصيلة استيعابك لهذه الوحدة مع الملخص التالي:

لقد حاولنا في هذه الوحدة تعريف الهياكل الشجرية وكذلك بيان المصطلحات المستخدمة والضرورية لعرض المواضيع الأخرى المتعلقة بها. ونظراً لأهمية الهياكل الشجرية الثنائية ولسهولة التعامل معها فقد تم التركيز على هذا النوع من الهياكل الشجرية.

وفي حديثنا عن الهياكل الشجرية الثنائية تناولنا طرق تمثيلها داخل ذاكرة الحاسوب والعمليات الأساسية التي يمكن إجراءها عليها، حيث درسنا طريقتين رئيستين لتمثيل الهياكل الشجرية الثنائية داخل ذاكرة الحاسوب هما:

1. التمثيل باستخدام المصفوفات ذات البعد الواحد (One dimensional array).
2. التمثيل باستخدام القوائم المتصلة (Linked list representation).

أما العمليات الأساسية على الهياكل الشجرية الثنائية، فتشمل العمليات الست التالية:

1. إنشاء الهيكل الشجري الثنائي (Binary Tree Creation)
2. استعراض الهيكل الشجري الثنائي (Binary Tree Traversal)
3. البحث عن عنصر معين من عناصر الهيكل الشجري الثنائي (Binary Tree Search)
4. إضافة عنصر إلى الهيكل الشجري الثنائي (Binary Tree Insertion)
5. حذف عنصر من الهيكل الشجري الثنائي (Binary Tree Deletion)
6. ترتيب القيم المخزنة في الهيكل الشجري الثنائي (Binary Tree sort)

وهناك عدة أنواع من الهياكل الشجرية الثنائية من أبرزها:
الكاملة، والعشوائية، والمتوازنة، والمقيدة. وقد تطرقنا لها بشــيء من التفصيل. كما تطرقنا إلى بعض التطبيقات العملية للهياكل الشجرية وخاصــة نوع B (B-Tree - B).

9. لمحة عن الوحدة الدراسية العاشرة

في الوحدة التالية، عزيزي الدارس، سنناقش نوعاً آخر من تراكيب البيانات غير الخطية، وهي المخططات والشبكات، وهي شبيهة بالهياكل الشجرية، إلا أنه يسمح لأي عنصر Node بالإشارة إلى أي عنصر آخر دون تقيد.

10. إجابات التدريبات

تدريب (1)

أ- A

ب- D, E

جـ- D, E, H, I

د- A

هـ- H, I, E, F, J

و- 4

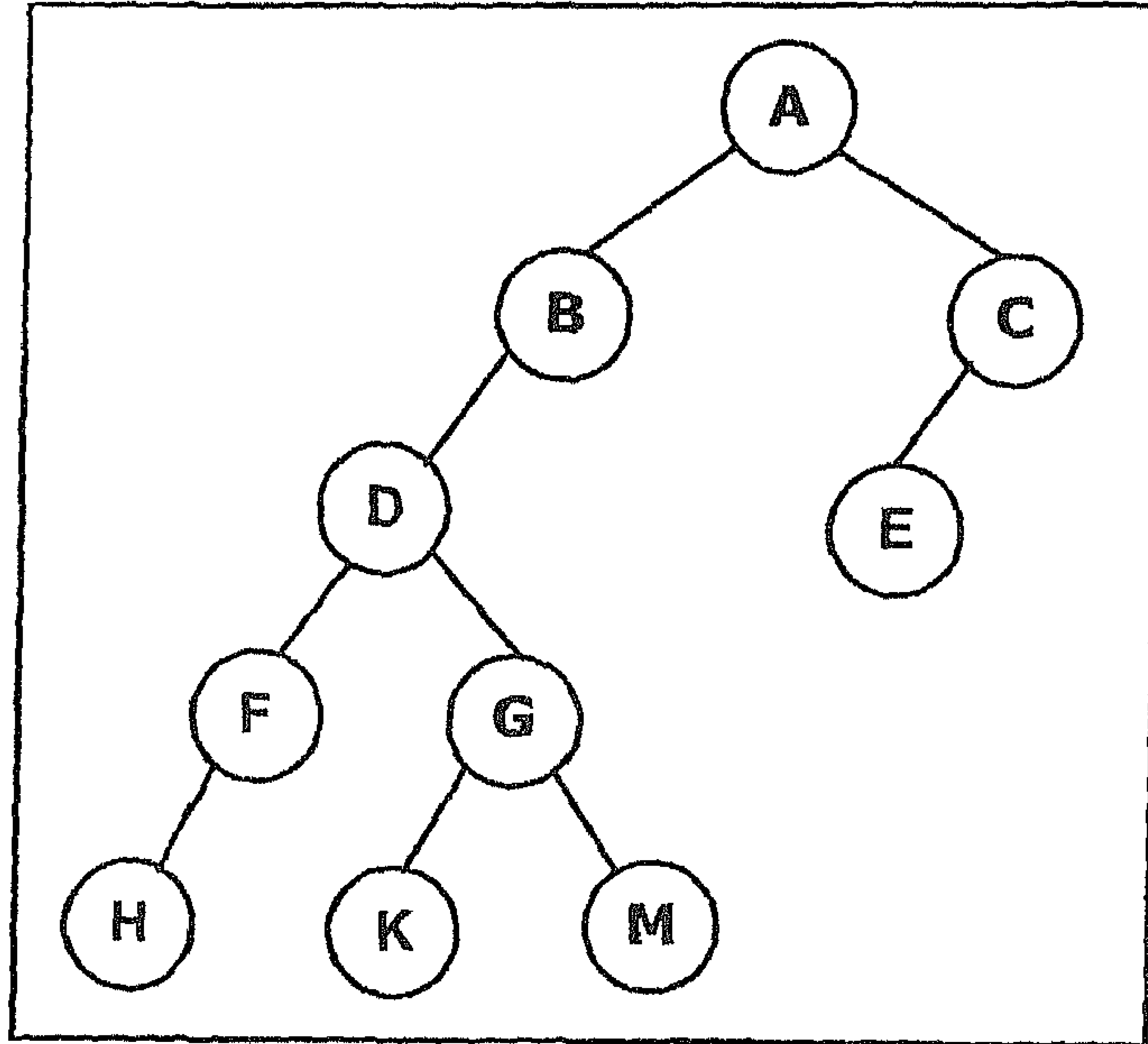
تدريب (2)

بما أن A هو جذر الشجرة، لذلك يتم تخزين الجذر في العنصر الأول من المصفوفة. وبما أن العنصر B يشكل الإبن الأيسر للعنصر A لذلك يحتل الموقع الثاني في المصفوفة أي (x12) حيث أن الموقع الأول هو موقع والد العنصر B. وبما أن العنصر C يشكل الابن الأيسر للموقع B، B يحتل الموقع الثاني، لذلك يمثل العنصر C في الموقع الرابع أي (x22). وهكذا.... إلى أن نحصل على التمثيل التالي للهيكل الشجري.

A	B		C				D								E
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

تدريب (3)

يمكن الحصول على ترتيب العناصر التي يتم طباعتها بالطريقة التالية وذلك برسم خطوط متقطعة حول الهيكل الشجري أولاً كما هو مبين أدناه.



من ثم يتم طباعة محتويات العنصر الذي نمر تحته وذلك بعد المرور على جميع عناصر الشجيرة اليسرى له، وبناءً على ذلك يتم استعراض العناصر وفق الترتيب التالي من اليسار إلى اليمين:

H F D K G M B A E C

تدريب (4)

يكون ترتيب العناصر من اليسار إلى اليمين هو:

A B C D F E G

ويتأتى هذا من فكرة الاستعراض وفق السياق الوسطي. حيث يتم طباعة العنصر بعد الانتهاء من زيارة جميع عناصر الهيكل الشجري الواقع في الجهة اليسرى من العنصر. وبما أنه لا توجد أبناء في الجهة اليسرى للعناصر A وB وC وD يتم طباعتها أولاً، ثم العناصر F وE وG .

تدريب (5)

الترتيب من اليسار إلى اليمين هو:

D C B A

وذلك بتطبيق الفكرة نفسها في المثال السابق حيث أنه يتم المسير إلى اليسار حتى لا نجد أبناء في الجهة اليسرى، لذلك يتم طباعة D أولاً ثم C ثم B ثم A .

تدريب (6)

الترتيب من اليسار إلى اليمين هو:

H F K M G D B E C A

ففي هذه الحالة يتم طباعة العنصر بعد زيارة جميع العناصر الواقعة في الهيكل الشجري الأيسر والأيمن لذلك العنصر. ولذلك يتم طباعة العنصر H أولاً لأنه لا يوجد له أبناء ومن ثم العنصر F والعنصر K وهكذا.

تدريب (7)

الترتيب من اليسار إلى اليمين هو:

F G E D C B A

وهذا التدريب مشابه للتدريب السابق باستثناء الاختلاف في الهيكل الشجري

تدريب (8)

الترتيب من اليسار إلى اليمين هو:

D C B A

وهذا التدريب أيضاً يشبه التدريبين السابقين باستثناء اختلاف الهيكل الشجري.

تدريب (9)

تكون نتيجة الطباعة من اليسار إلى اليمين كما يلي:

A B D F H G K M C E

وهذه النتيجة تنبع من أنه في الاستعراض وفق السياق القبلي يبدأ الاستعراض من الجذر ويتم طباعته. بعد ذلك يتم المسير باتجاه الابن الأيسر وهكذا حتى لا يتبقى ابن أيسر للعناصر ولذلك يتم طباعة العناصر A B D F H. عندئذ يتم التراجع إلى النقطة D ونبدأ المسير بانسحاب الابن الأيمن ويتم طباعة العنصر G . ثم يتم المسير بعدئذ باتجاه الابن الأيسر للعنصر G، ويتم طباعة ECMK .

تدريب (10)

تكون نتيجة الطباعة من اليسار إلى اليمين كما يلي:

A B C D F E G

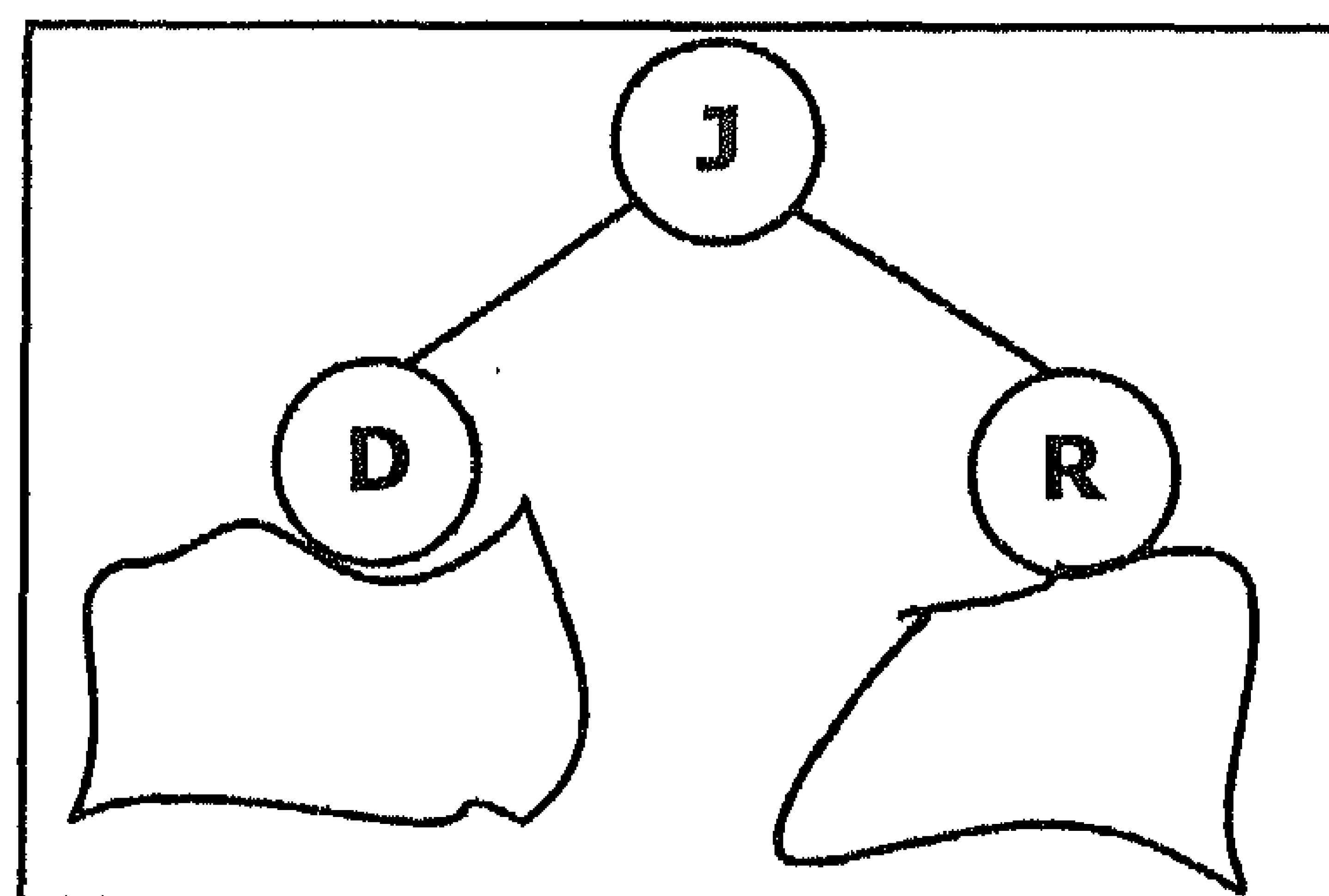
تدريب (11)

تكون نتيجة الطباعة من اليسار إلى اليمين كما يلي:

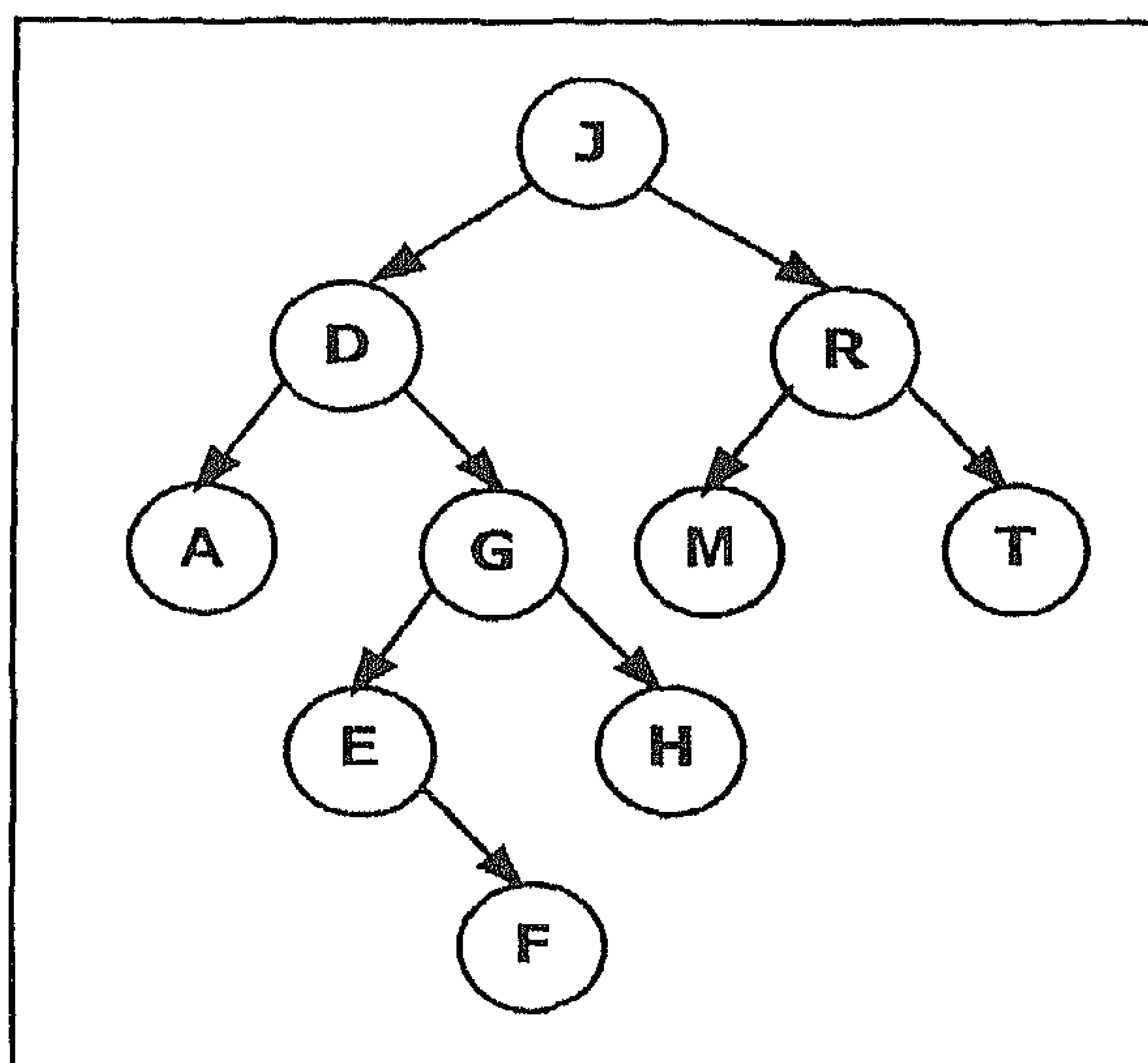
A B C D

تدريب (12)

ننظر أولاً إلى نتيجة الاستعراض وفق السياق القبلي. وبما أن العنصر الأول نتيجة لهذا الاستعراض هو J، لذلك فإن J تشكل جذر الهيكل الشجري. بعد ذلك نرجع إلى الاستعراض وفق السياق الوسطي، ونجد أن جميع العناصر الواقعة إلى يسار العنصر J يجب أن تكون في الهيكل الشجري الأيسر للعنصر J وهذه العناصر هي (A D E F G H) . وأن العناصر الواقعة إلى يمين العنصر J (وهي العناصر M R T) بعد ذلك نرجع ثانية إلى الاستعراض وفق السياق القبلي، فنلاحظ أن العنصر D هو العنصر الأول من قائمة العناصر المشكلة للهيكل الشجري الأيسر للعنصر J وكذلك فإن العنصر R هو العنصر الأول من قائمة العناصر المشكلة للهيكل الشجري الأيمن للعنصر J . وبذلك يشكل العنصرين الجذرين للهيكل الشجري الأيسر والأيمن للعنصر J كما يلي:

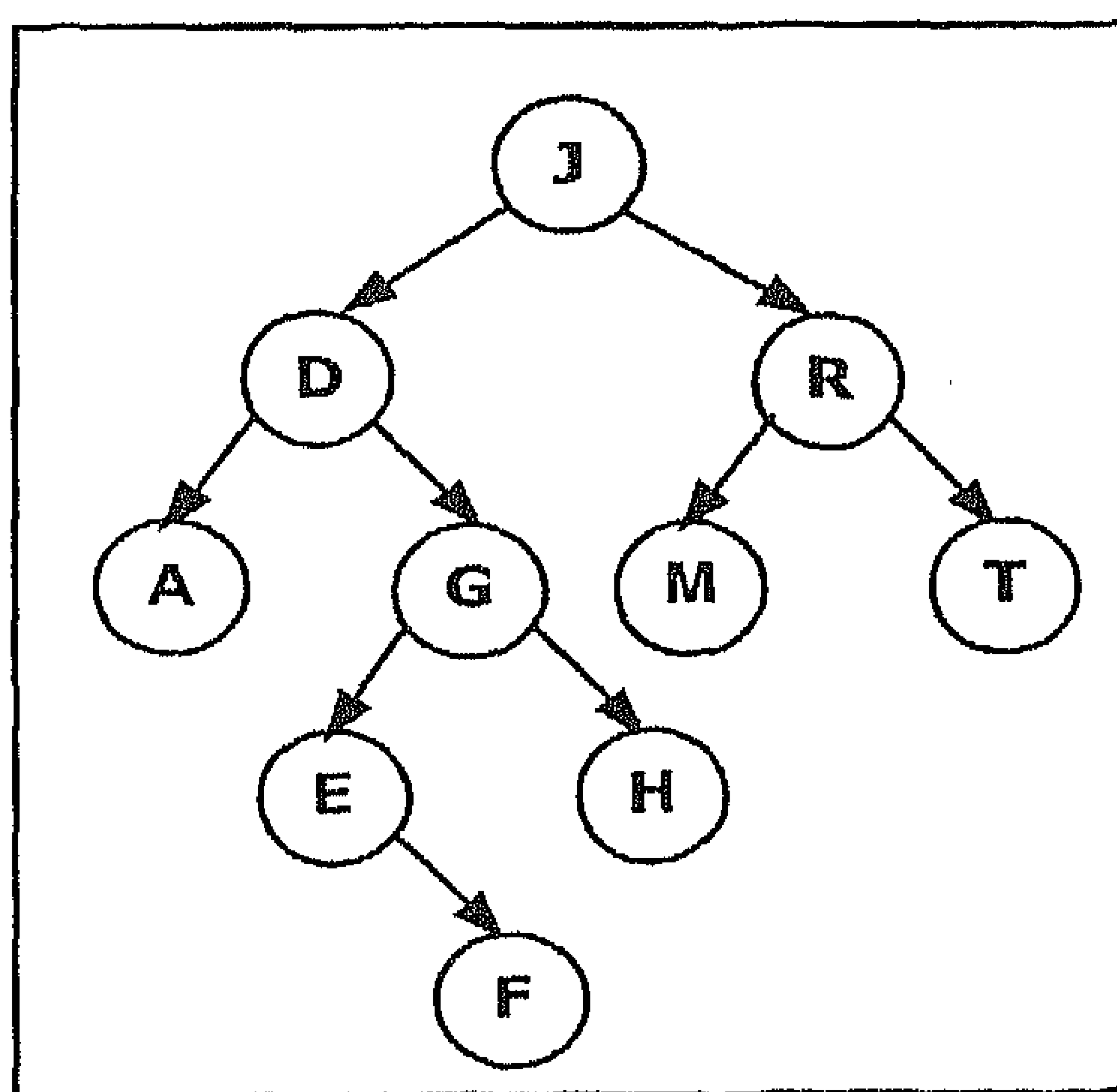


مرة أخرى نعود إلى نتيجة الاستعراض وفق السياق الوسطي فنجد أن جميع العناصر الواقعة إلى يسار العنصر D (وهو العنصر A).



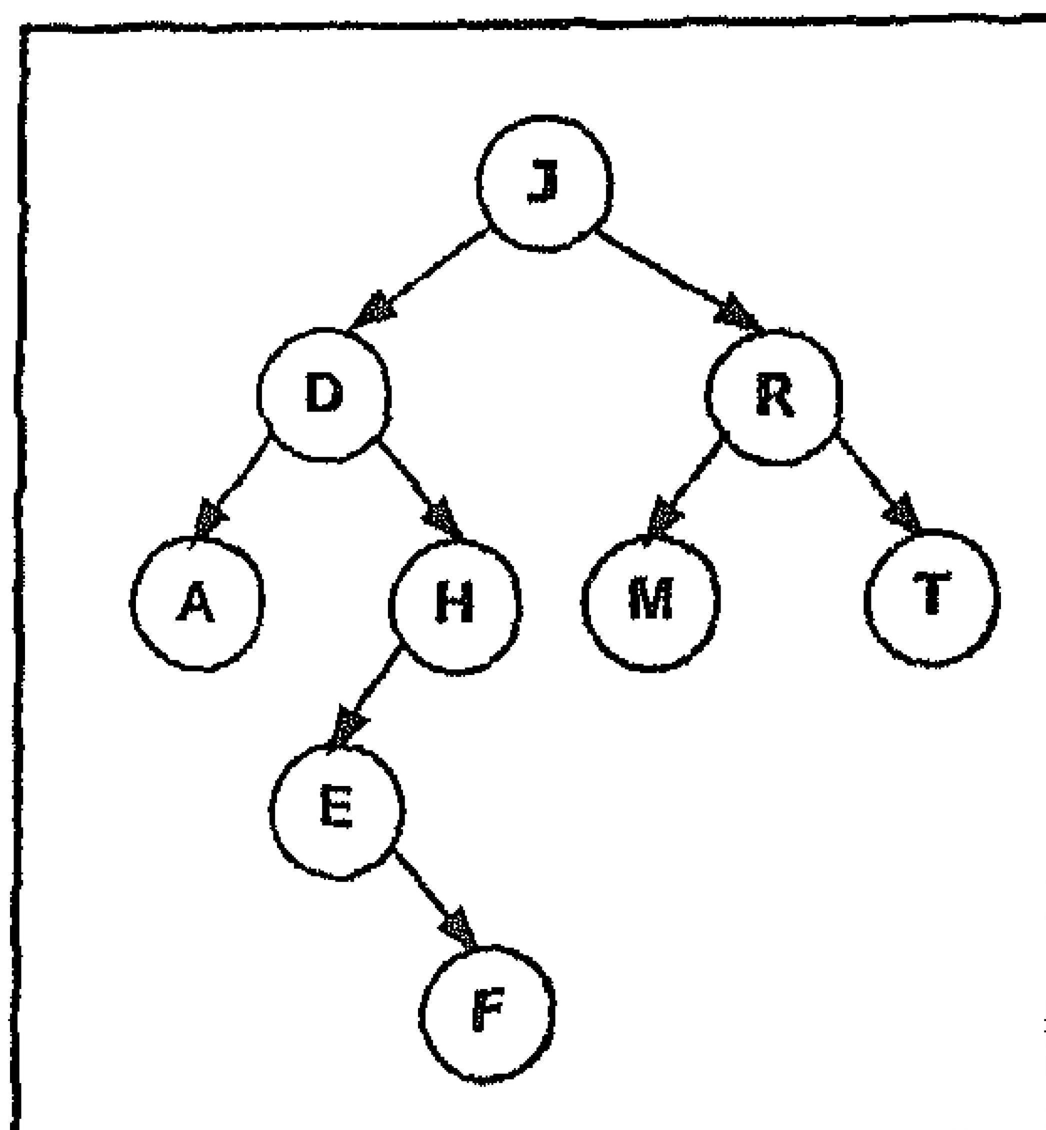
تدريب (13)

بما أن أول عنصر هو J، لذلك فإن موقع هذا العنصر هو جذر الهيكل الشجري. أما العنصر الثاني فهو R . وبما أن R يأتي بعد الحرف J في الأحرف الأبجدية، لذلك يجب أن يكون الابن الأيمن للعنصر J . أما العنصر الثالث وهو D يجب أن يكون في موقع الابن الأيسر للعنصر J لأنه يقع قبل العنصر J في ترتيب الأحرف الأبجدية. بعد ذلك نرغب في إدخال العنصر G إلى الهيكل الشجري الثاني. وبما أن G أصغر من J يكون في الشجيرة اليسرى للجذر J . وبما أن G أكبر من D لذلك فإن G تقع في موقع الابن الأيمن للعنصر D ونستمر بهذه الطريقة حتى يتم بناء الهيكل الشجري.

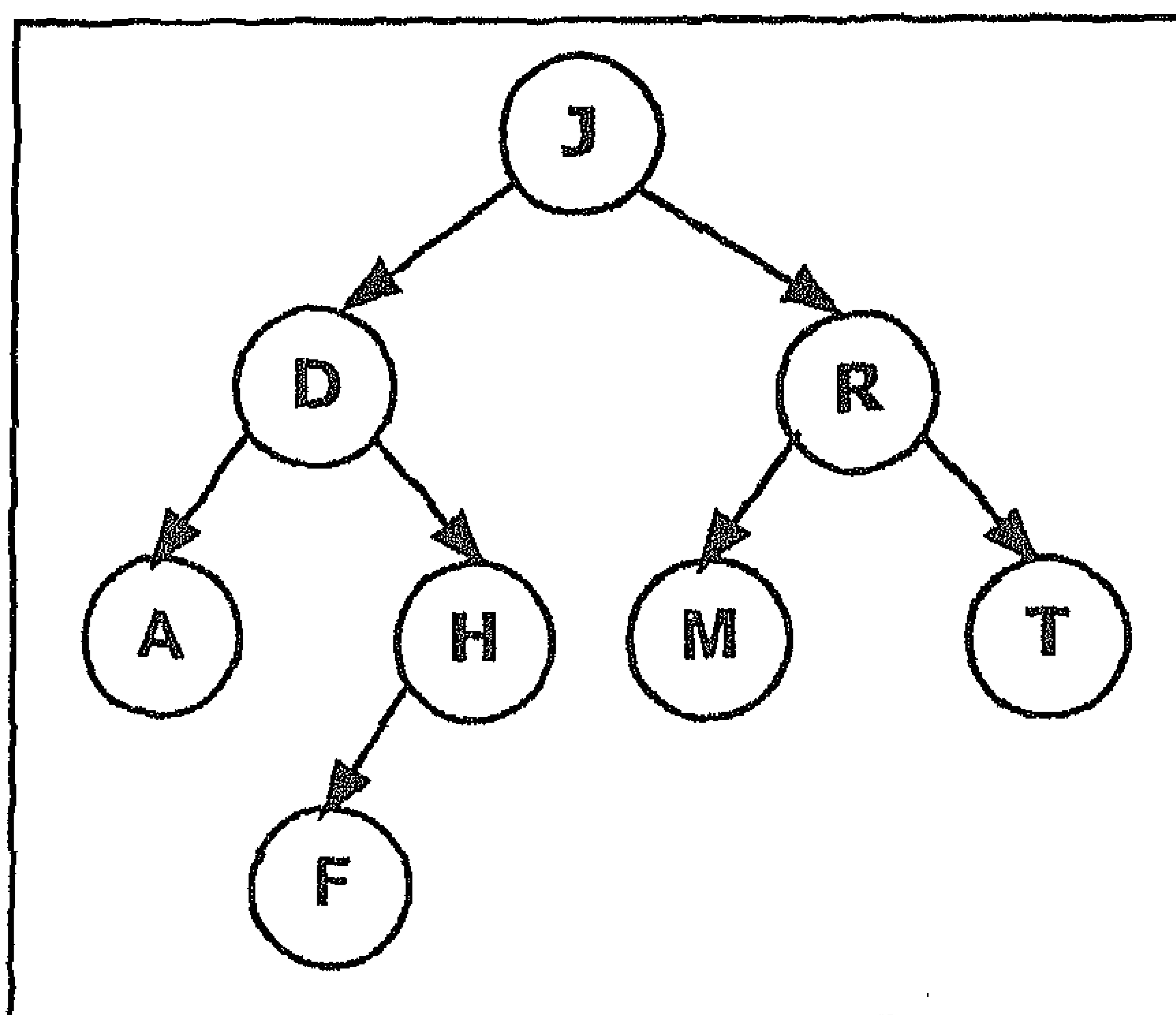


تدريب (14)

أ- عند حذف العنصر G نلاحظ أن هناك ابنان اثنان للعنصر G ولذلك يتم استبدال G بالعنصر الأكبر منهما وهو H ولذلك يصبح E على أنه الابن الأيسر للعنصر H. وهكذا نحصل على الهيكل الشجري التالي:



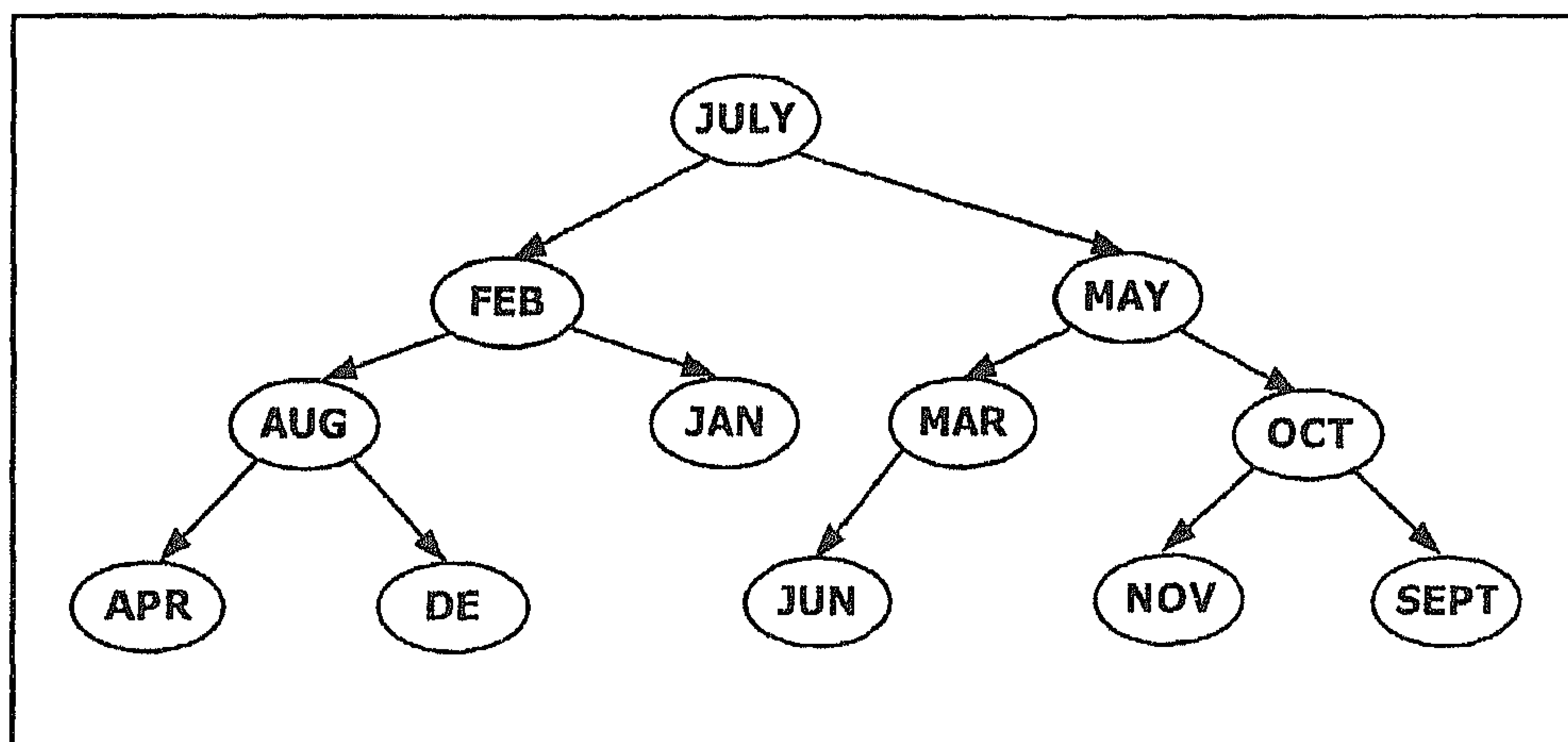
ب- أما عند حذف العنصر E من الهيكل الشجري الجديد يصبح F الابن الأيسر للعنصر H كما هو مبين في الشكل التالي:



تدريب (15)

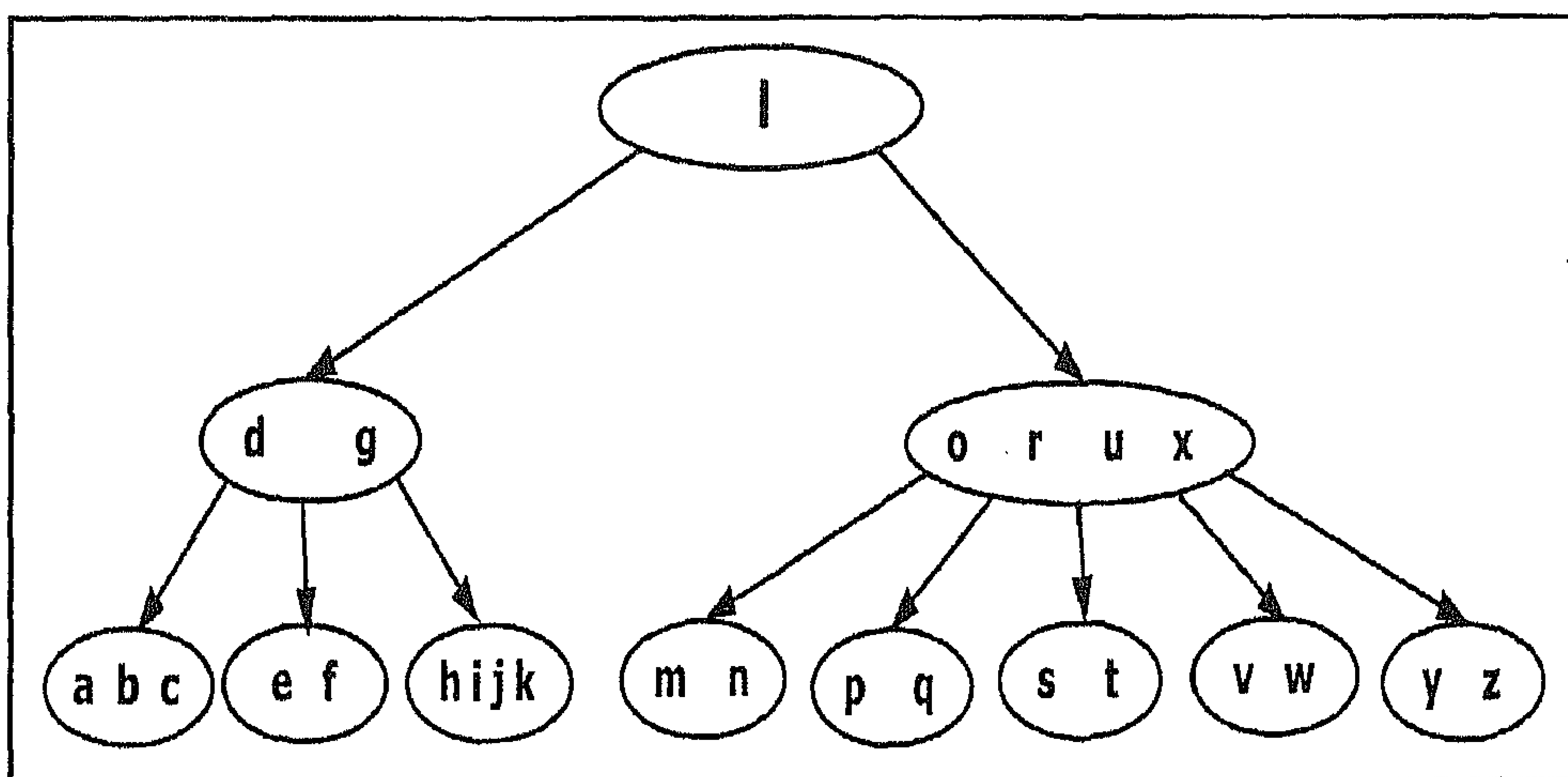
بالرجوع إلى أشهر السنة يجب ملاحظة أن الشهر July يتوسط أشهر السنة من حيث ترتيب الكلمات الهجائي حيث نجد أن هناك خمسة أشهر تقع قبل July في الترتيب وأن هناك ستة أشهر تقع بعد July من حيث الترتيب، لذا فإنه من الحكمة أخذ الشهر July ليمثل جذر الهيكل الشجري. وبالطريقة ذاتها نحاول اختيار موقع العناصر المحافظة على

الهيكل الشجري لكي يبقى متوازناً. فعلى سبيل المثال، لو استبدلنا FEB بـ DEC نلاحظ أن الهيكل الشجري يصبح غير متوازن. ومن الممكن الحصول على أكثر من هيكل شجري متوازن لأشهر السنة والشكل المبين هو أحدها:



تدريب (16)

هذا هو أحد الحلول الممكنة لهذا التدريب. ورتبة هذا الهيكل الشجري هو 5 لأنه يحتوي على بعض المواقع التي تحتوي على أربعة عناصر مثل الموقع الذي يحتوي على الأحرف (O r u x). أما عمق هذا الهيكل الشجري فهو 3 كما هو مبين. وتجدر الإشارة إلى أنه يمكن الحصول على هياكل شجرية أخرى من النوع-B وتحمل الرتبة 5 والعمق 3.



11. مسرد المصطلحات

- **الاستعراض وفق السياق التبعي Traversal Postorder**: نوع من أنواع الطرق التكرارية لاستعراض الهيكل الشجري الثنائي بحيث يتم معالجة أي من العناصر بعد زيارة أبناء ذلك العنصر.
- **الاستعراض وفق السياق القبلي Preorder Traversal**: نوع من طرق استعراض الهيكل الشجري بحيث يتم معالجة (أو زيارة) العنصر قبل أبنائه. ويعتمد هذا النوع من طرق الاستعراض على مبدأ الاستدعاء الذاتي.
- **الاستعراض وفق السياق الوسطي Inorder Traversal**: نوع من طرق الاستعراض للهيكل الشجري الثنائي بحيث يتم زيارة الابن الأيسر لأي من العناصر أولاً قبل طباعة ذلك العنصر (أي معالجته). وبذلك يكون الترتيب على النحو التالي وبشكل تكراري Recursive: سر باتجاه الابن الأيسر ثم اطبع العنصر وأخيراً سر إلى الابن الأيمن.
- **نظام الرموز التبعي Postfix Notation**: في هذا النظام تكون العملية الحسابية الثنائية مباشرة بعد المعاملين الخاصين بها. ولذلك لا توجد حاجة لاستخدام الأقواس في هذا النظام. فعلى سبيل المثال فإن التعبير +AB باستخدام نظام الرموز التبعي يكافيء التعبير A + B في نظام الرموز الوسطي.
- **نظام الرموز القبلي Prefix Notation**: يطلق على التعابير الحسابية التي تسبق فيها العملية الحسابية ثنائية المعاملين الخاصين بها. فعلى سبيل المثال +AB تعني A+B في النظام المتعارف عليه والذي يطلق عليه نظام الرموز الوسطي. وفي هذا النظام لا تستخدم الأقواس.
- **نظام الرموز الوسطي Infix Notation**: يطلق على النظام الذي تكون فيه العملية الحسابية الثنائية بين معامليها. وفي هذا النظام يمكن استخدام الأقواس للتحكم في ترتيب تنفيذ العمليات.
- **الهياكل الشجرية Trees**: نوع من تراكيب البيانات الهرمية بحيث ترتبط العناصر معاً بعلاقة الوالد والأبناء. فهناك والد واحد لأي من الأبناء، ويمكن للوالد الواحد أن يكون له أي عدد من الأبناء.
- **الهياكل الشجرية الثنائية Binary Trees**: نوع من الهياكل الشجرية بحيث لا يزيد عدد الأبناء لأي من العناصر عن ابنين اثنين.

- **الهياكل الشجرية الثنائية العشوائية Random Binary Trees**: نوع من أنواع الهياكل الشجرية الثنائية تكون العناصر فيه عشوائية ولا تخضع لأي ترتيب وينتج عن ذلك أن الهيكل الشجري لا يكون متوازناً من حيث العمق.
- **الهياكل الشجرية الثنائية الكاملة Complete Binary Trees**: نوع من الهياكل الشجرية الثنائية تحتوي فيه جميع المستويات في الهيكل الشجري (مع احتمال استثناء المستوى الأخير) على أكبر عدد ممكن من العناصر المسموح به (أي أن المستوى i، مع احتمال استثناء المستوى الأخير يحتوي على $2i + 1$ من العناصر. وكذلك فإن العناصر على المستوى الأخير تظهر من أقصى اليسار إلى أقصى اليمين.
- **الهياكل الشجرية الثنائية المتوازنة Balanced Binary Trees**: نوع من الهياكل الشجرية الثنائية تكون فيه جميع العناصر على المستوى نفسه. كما أن عدد الأبناء المنبثق من أي من العناصر إما أن يكون اثنان أو لا شيء.
- **الهياكل الشجرية الثنائية المقيدة Heaps**: نوع من الهياكل الشجرية الثنائية الكاملة يتميز بأن القيمة المخزنة في أي موقع من المواقع يجب أن تكون مساوية أو أكبر من القيم المخزنة في العنصرين اللذين يمثلان ابني العنصر في ذلك الموقع إذا كان لهذا العنصر أبناء. ولذلك تكون القيمة المخزنة في الجذر أكبر القيم في الهيكل الشجري المقيد.



12. المراجع

1. Clifford A. Shaffer, Practical Introduction to Data Structures and Algorithm Analysis (C++ Edition), 2nd Edition, Prentice -Hall. 2000.
2. Tremplay, J.P.; and Sorenson, P.G.; An Introduction to Data Structures with Applications, 2nd Edition., McGraw-Hill, 1984.
3. Amsbury, Wagne, Data Structures from Arrays to Priority Queues. Belmont (USA): Wadsworth, 1985.
4. Kruse, Robert L., Data Structures and Program Design. Englewood Cliffs (USA): Prentice-Hall, 1984.
5. Lipschutz, Seymour, Theory and Problems of Data Structures. New York: McGraw-Hill, 1986.
6. Miller, Lawrence H., Advanced Programming Design and Structures. New York: McGraw-Hill, 1986.
7. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison Wesley, 1999.
8. Lewis, T. G.; Smith, M.Z., Applying Data Structures. Atlanta (USA): Houghton Mifflin, 1976.
9. Tenenbaum, Aarom M.; Augenstein, Moshe J., Data Structures Using Pascal. Englewood Cliffs (USA): Prentice-Hall, 1981.



المخططات والشبكات
(Graphs & Networks)

محتويات الوحدة

1. المقدمة

1.1 تمهيد

أهــلاً بــك، عزيزي الدارس، إلى الوحدة العاشــرة مــن كتاب «تركيــب البيانات وتصميم الخوارزميات» وهي بعنوان «المخططات والشبكات».

فــي هذه الوحــدة، عزيزي الــدارس، ســنناقش أيضاً تراكيــب بيانات غيــر خطية وهي المخططات والشــبكات اللتان تســتخدمان في تمثيــل العديد من البيانات المســتخدمة في الحياة اليومية مثل شــبكات الطرق بين المدن أو شبكات الهاتف ورحلات الطيران المنظمة بين المدن وما إلى ذلك. ســتدرس في هذه الوحدة طرق تمثيل المخططات والشبكات كذلك أهم الخوارزميات الخاصة بهذه التراكيب البيانية.

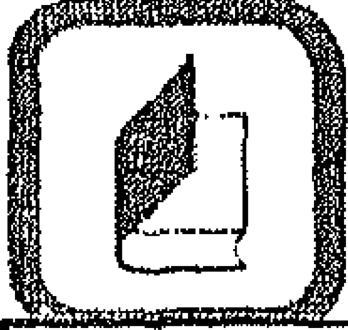
2.1 أهداف الوحدة

ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1- تُعرف مفهوم المخططات والشبكات وتميزّ بينها.
2- تمثل المخططات والشبكات بطرق مختلفة.
3- تستطيع استخدام خوارزمية Dijkstra لإيجاد أقصر ممر بين زوجين من العناصر.
4- تستطيع استخدام خوارزمية Floyd لإيجاد اقصر ممر ما بين كل زوجين من العناصر.
5- تميّز بين الهياكل الشجرية من جهة والمخططات والشبكات من جهة أخرى.
6- تبيّن أهم التطبيقات العملية للمخططات.

3.1 أقسام الوحدة

تقســم هذه الوحدة إلى أربعة أقســام رئيســة ترتبط بقائمة الأهداف السابقة. ففي القسم الأول ســتتعرف، عزيزي الدارس، على مفهوم المخططات وطرق تمثيلها في ذاكرة الحاسوب، ويرتبط هذا القســم بالهدفين الأول والثاني. أما في القســم الثاني فسنناقش مفهوم الشبكات وطــرق تمثيلها وخوارزميتي Dijkstra و Floyd لإيجاد أقصر الممرات، ويرتبط هذا القســم بتحقيــق الأهــداف الأول والثاني والثالــث والرابع. وفي القســم الثالث ســنناقش العلاقة بين الهياكل الشــجرية والمخططات، ويرتبط هذا القسم بالهدف الخامس. أما في القسم الخامس، وهو الأخير، فســنقدم أمثلة على تطبيقات المخططات والشبكات، ويرتبط هذا القسم بالهدف السادس.



4.1 القراءات المساعدة

على الرغم من شـمولية الأقسـام والمسـائل التي تم عرضها في هذه الوحدة إلا أن هناك فوائـد كثيرة يمكـن أن تجنيها عزيزي الـدارس، من الرجوع إلى بعض القـراءات الإضافية المسـاعدة. فهناك المزيد من المعلومات والأمثلـة في هذه المصادر، وهناك أيضاً المزيد من التدريبات والأسئلة. ونوصي بالمصدرين التاليين لهذه الغاية، مع التذكير بأن قائمة المراجع في نهاية هذه الوحدة تتضمن مصادر على قدر كبير من الأهمية والفائدة.

1. Amsbury, Wayne, Data Structures from Arrays to Priority Queues. Belmont (USA): Wadsworth, 1985. pp. 97-119, 120-141, & 169-198.
2. Miller, Lawrence H., Advanced Programming Design and Structures. New York: McGraw-Hill, 1986.

2. المخططات Graphs

تحدثنا في الوحدة السابقة، عزيزي الدارس، عن الهياكل الشجرية والتي تعتبر نوعاً خاصاً من المخططات والتي تستخدم لتمثيل بعض العلاقات الهرمية بين البيانات. وفي هذه الوحدة نتحدث عن المخططات والشبكات لتمثيل البيانات. وتمتاز المخططات بقلة القيود المفروضة عليها مقارنة مع الهياكل الشجرية، حيث أنه من الممكن أن يؤشر أي عنصر على أي عنصر آخر فيها. وهذا يعني أن علاقة الأب والابن غير موجودة في حالة المخططات.

يتكون المخطط الواحد من جزئين رئيسين هما:

1. مجموعة من العناصر (Nodes أو Elements) يطلق عليها في بعض الأحيان رؤوساً (Vertices)، وسنرمز إليها بالرمز V.
2. مجموعة من الحواف (Edges) وتربط كل حافة عنصرين . وتتميز كل حافة من هذه الحواف بزوج مميز من عناصر المجموعة V. وسنرمز إلى مجموعة الحواف بالرمز E.

وعلى هذا الأساس يشار إلى المخطط G كما يلي:

$$G= (V , E)$$

حيث تمثل V مجموعة العناصر أو الرؤوس، وتمثل E مجموعة الحواف. ومجموعة العناصر هذه تمثل البيانات المخزنة في المخطط. وأما الحواف فتمثل الممر من عنصر إلى آخر وذلك للربط بين العناصر المختلفة.

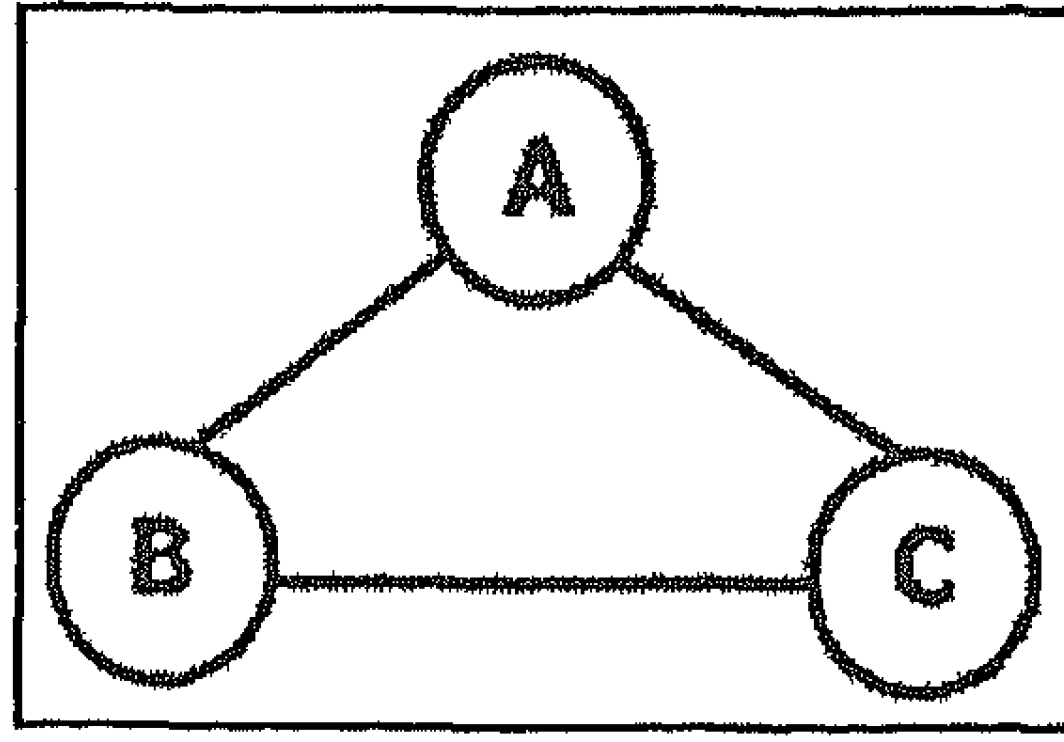
1.2 أنواع المخططات

يوجد نوعان رئيسان من المخططات هما:

1 - المخطط غير المتجه (Undirected Graph)

يمثل الشكل (1) مخططاً غير متجه يتكون من مجموعة العناصر (A, B, C) ومن مجموعة الحواف {(A, B), (B, C) , (A, C)}

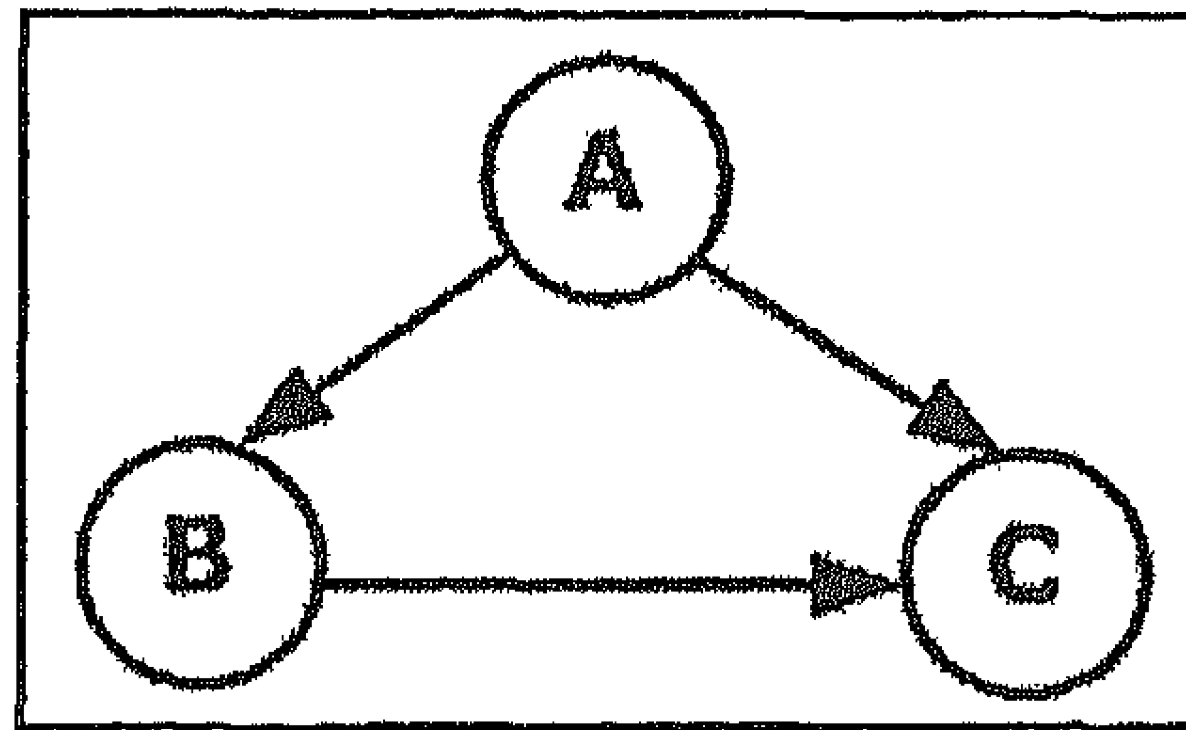
حيث تعني الحافة (A, B) إن العنصر A مرتبط بالعنصر B وبنفس الطريقة التي يرتبط بها العنصر B بالعنصر A كأن الحافة (A, B) هي طريق باتجاهين) وعليه فإن الحافة (A, B) تعني أيضاً (B, A) وهاتان الحافتان متكافئتان ولا حاجة للتكرار، ويكفي ذكر واحدة منهما.



شكل (1)

2 - المخطط المتجه (Directed Graph) أو (Digraph)

يمتاز هذا النوع من المخططات بوجود اتجاهات معينة للحواف يشار إليها بعلامة الاتجاه. وفي هذه الحالة فإن الحافة (A, B) غير مكافئة للحافة (B, A) (وكأن الحافة المتجهة طريق باتجاه واحد). فعلى سبيل المثال، في المخطط المبين في الشكل (2) تعتبر (A, B) من ضمن حواف المخطط بينما الحافة (B, A) ليست من حواف المخطط، وذلك لعدم وجود خط متجه من العنصر B إلى العنصر A، بينما يوجد خط متجه من العنصر A إلى العنصر B.



شكل (2)

وتبرز أهمية المخطط المتجه في تطبيقات عدة في حياتنا اليومية. فعلى سبيل المثال يمكن تمثيل شبكة ضخ البترول على شكل مخطط، تمثّل محطات الضخ بعناصر (أو رؤوس) المخطط، وتمثل خطوط ضخ النفط بالحواف، بحيث أنه إذا أردنا تمثيل خط نقل نفط من المحطة (A) إلى محطة (B) نرسم خطاً متجهاً من المحطة (A) إلى المحطة (B). كذلك خطوط الهاتف والكهرباء يمكن تمثيلها كمخططات.

وسنركز، عزيزي الدارس، في الأقسام التالية على المخططات المتجهة.

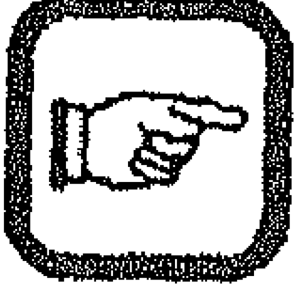
2.2 طرق تمثيل المخططات

هناك طريقتان رئيستان لتمثيل المخططات هما: استخدام القوائم المتصلة (Linked Representation) واستخدام مصفوفة الجوار (Adjacency Matrix)

1.2.2 تمثيل المخططات باستخدام مؤشرات الربط

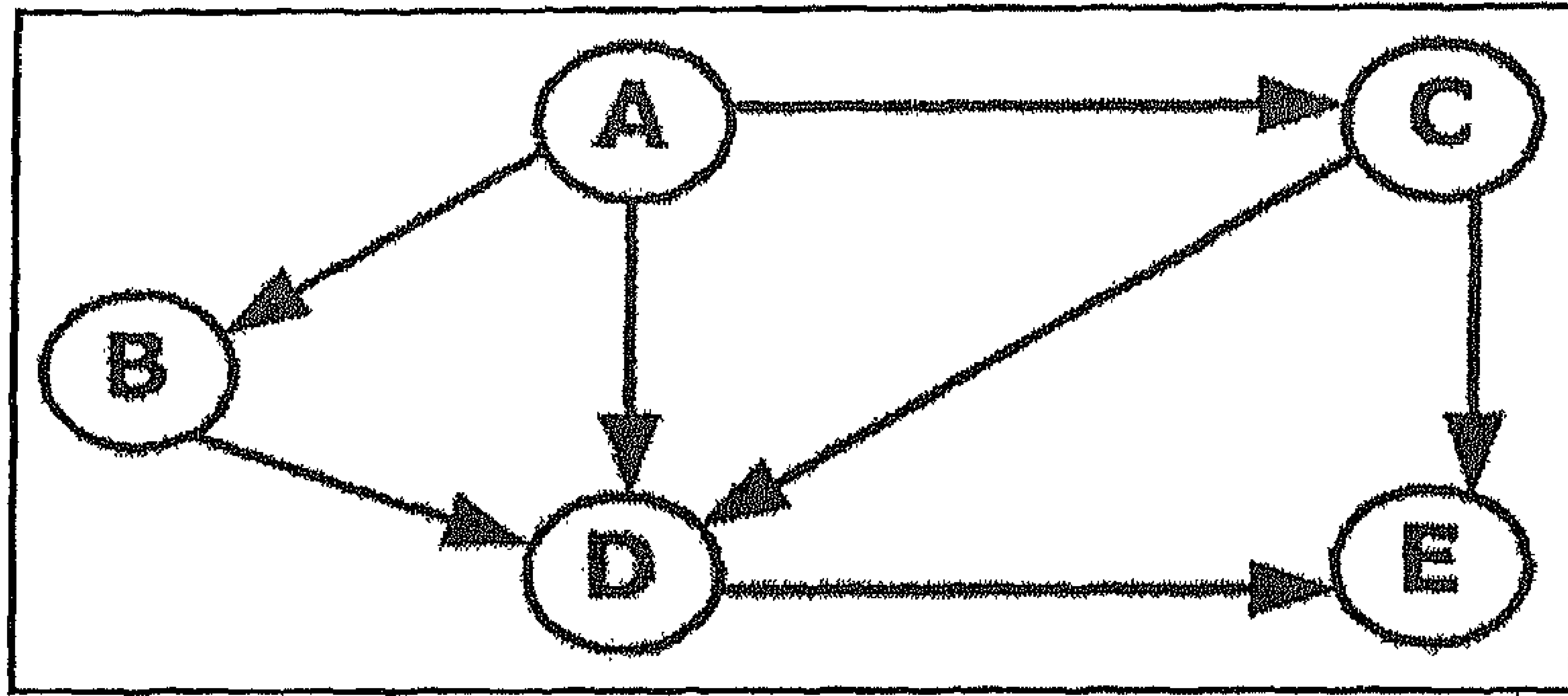
(Linked Representation)

فـي هذه الحالة، وبما أنه من الممكـن أن يرتبط العنصر مع عدد غير محدود من العناصر الأخرى، فلا شـك بأن طريقة التمثيل سـوف تختلف اختلافاً جذرياً عن طريقة تمثيل الهياكل الشجرية الثنائية. وسنوضح هذه الطريقة بالمثال التالي:



مثال (1)

افرض أن لدينا المخطط المبين في الشكل (3)، ونود تمثيله باستخدام هذه الطريقة.

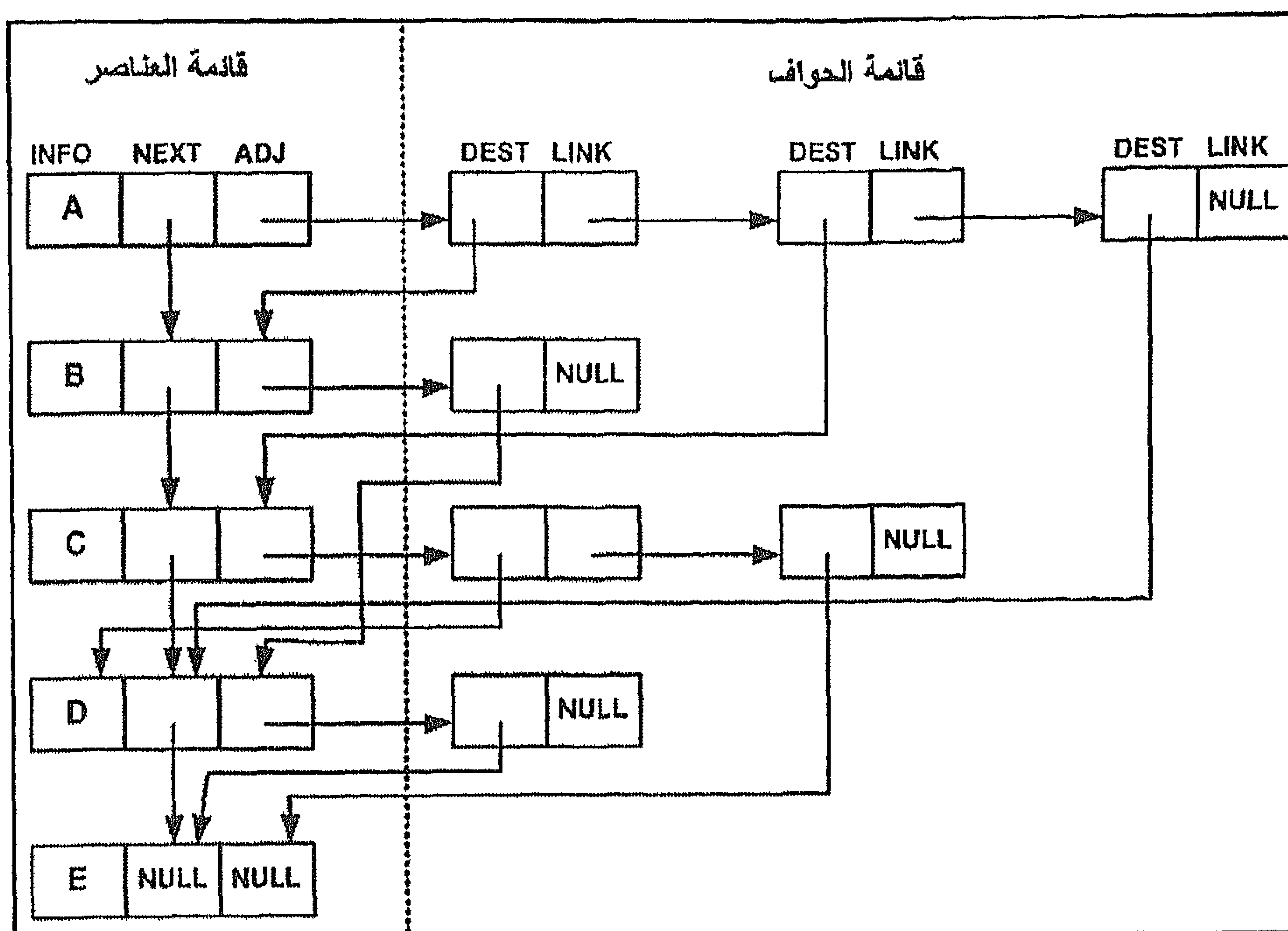


شكل (3)

فـأول ما نقوم بـه هو ترتيب عناصر المخطـط ترتيباً عمودياً، حيـث يمثل كل عنصر من العناصـر بثلاثة حقول، حقل للبيانات، وحقل يربط هذا العنصر بالعنصر التالي، وحقل ثالث يربـط هذا العنصر مع أول عنصر من قائمة العناصـر التي ترتبط معه (أي التي تربطها بهذا العنصر حافة متجهة).

وكمـا نلاحظ من الشـكل (4) فإن عناصـر المخطط ترتبط مع بعضها البعض بواسـطة الحقل المسمى (NEXT). أما الحقل (ADJ) فيستخدم للتأشير على قائمة الحواف (Edges) فـي المخطط، والتي تبدأ من العنصر الذي يحتوي على هذا الحقل كما هو واضح في الشـكل (4)، والذي نلاحظ من خلاله أن تمثيل المخطط بهذا الشكل يشمل قائمتين هما:

1 - قائمة العناصر: وتتكون من العناصر الخمسة إلى يسار الخط المتقطع.



شكل (4)

2- **قائمة الحواف:** وتتكون من العناصر الواردة إلى يمين الخط المتقطع، ويمثل كل عنصر من هذه العناصر إحدى الحواف. ويتكون كل عنصر من هذه القائمة من حقلين. يمثل الحقل الأول، والذي أشرنا إليه بالاسم (DEST) مؤشراً إلى العنصر الآخر من طرفي الحافة. بالرجوع إلى الشكل (4) وخاصة إلى قائمة حواف العنصر A (أي السطر الأول من قائمة الحواف)، نجد أن المؤشر (DEST) لأول حافة من الحواف يؤشر على العنصر (B) من قائمة العناصر، وهذا يعني أن هذه الحافة هي (A, B)، أما المؤشر (DEST) للحافة الثانية فيتجه إلى العنصر (C)، وهذا يعني أن هذه هي الحافة (A, C) وهكذا. أما الحقل الثاني فيشير إلى الحافة التالية التي تربط العنصر مع عنصر آخر يتم تحديده بواسطة المؤشر (DEST).

ويمكننا تعريف هذا التمثيل في لغة سي++ كما يلي:

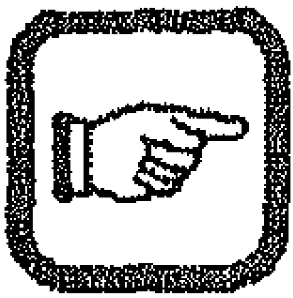
```
typedef char InfoType;
typedef struct ElementNode
{InfoType INFO;
```

```
 ElementNode* NEXT;
 EdgePtr*    ADJ;
}ElementNode;

typedef struct EdgePtr
 {ElementNode* DEST;
 EdgePtr*    LINK;
}EdgeNode;
```

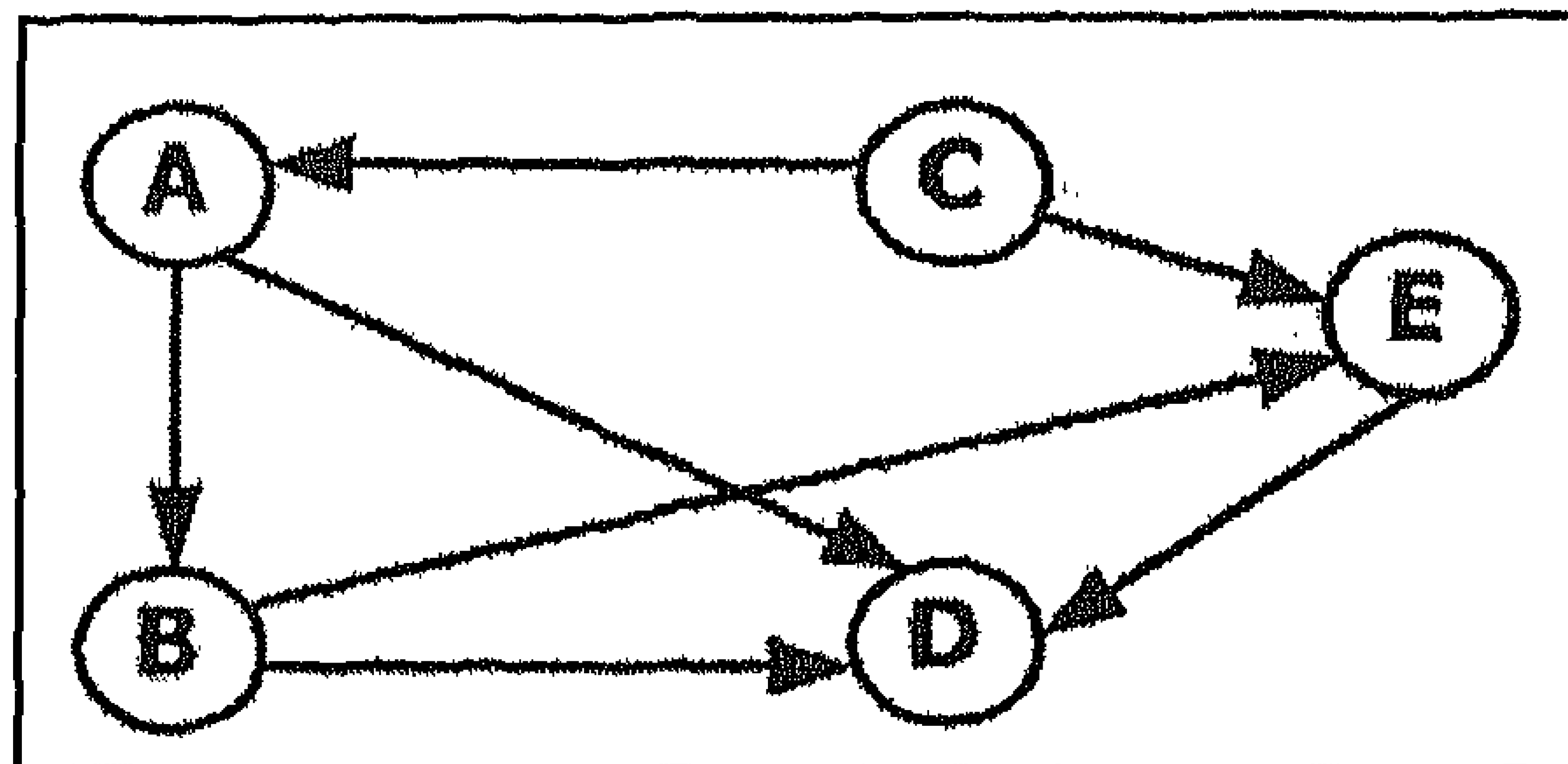
حيث أن:

(INFO) يمثـل البيانات أو القيـم الموجودة في العناصر ويمكـن أن تحتوي على بيانات مـن الأنواع البسـيطة مثل الصحيحـة والحقيقية والرمزيـة أو الأنواع المركبة مثل السـجلات والمصفوفـات وسلاسـل الرموز وما إلى ذلـك. ويمكن كذلك إضافة حقل ثالـث إلى كل حافة يحتوي على تكلفة أو وزن الحافة حيث يمكن اسـتعمال هذا الحقل عند إيجاد أقصر الممرات باستخدام المخططات والذي يشكل موضوع الجزء التالي.



مثال (2)

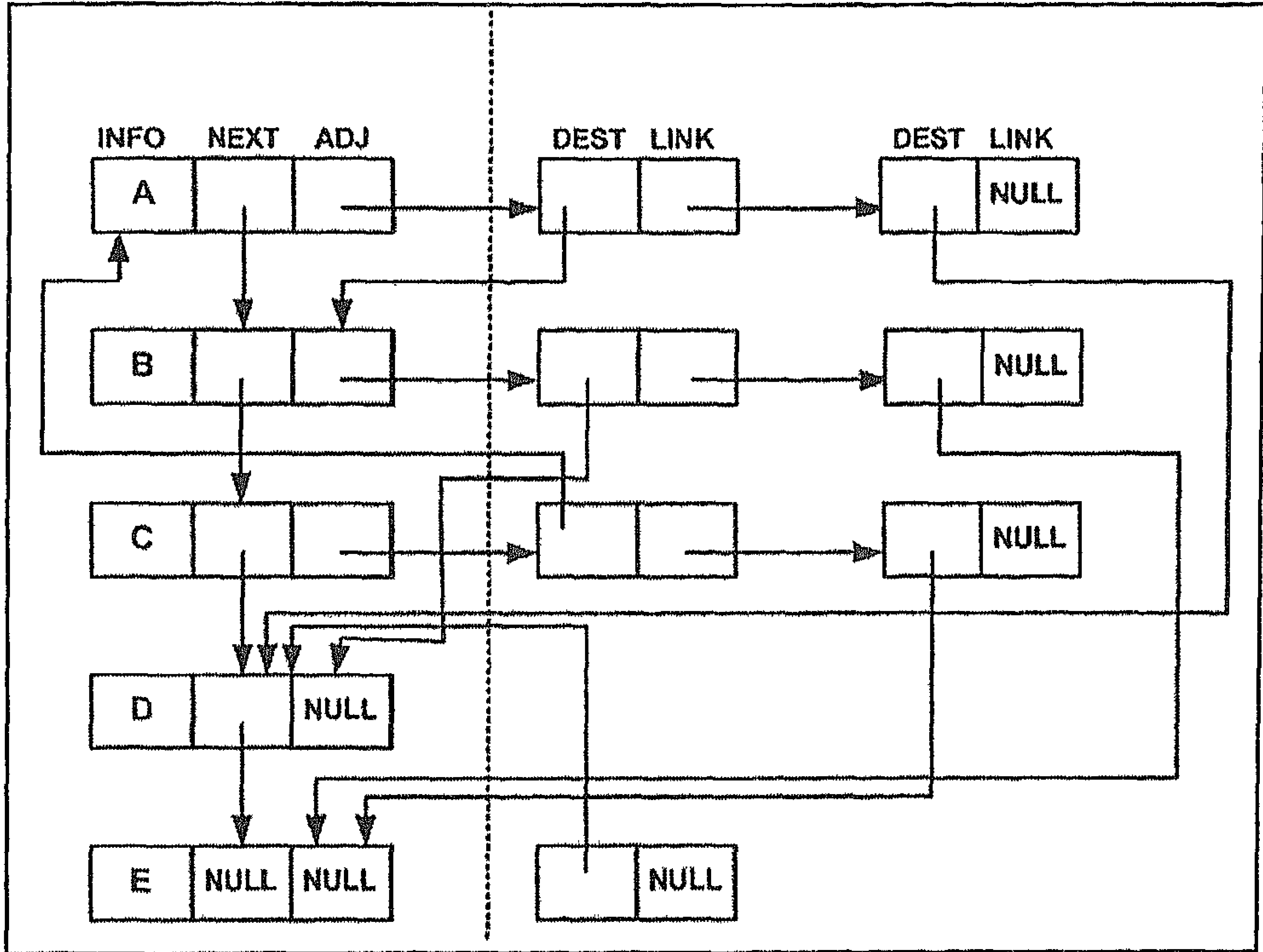
مثّـل المخطـط المبيـن فـي الشـكل (5) باسـتخدام مؤشـرات الربـط (Linked Representation)



شكل (5)

الحل:

كما هو مبين في الشكل (6):

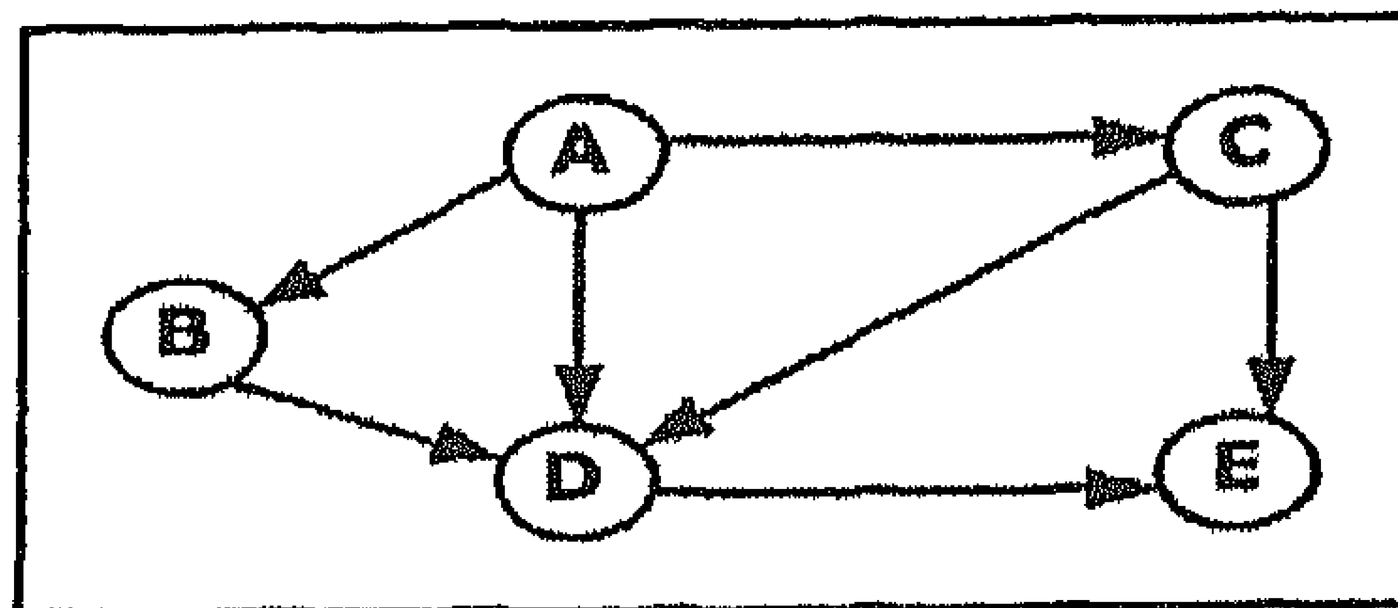


شكل (6)

وللتأكد من أن هذا التمثيل هو تمثيل صحيح، يمكن أخذ العناصر عنصراً عنصراً. فلو بدأنا بالعنصر (A) نجد أن الحقل (ADJ) يؤدي إلى العنصر (B) أي أن الحافة المتجه (A, B) هي من حواف المخطط. وهذا صحيح كما هو واضح في الشكل (5). وبتتبع الحقل (LINK) لهذه الحافة نصل إلى الحافة (A, D) وهذه أيضاً إحدى حواف المخطط. وبما أن قيمة الحقل (LINK) لهذه الحافة يساوي (NULL) فهذا يعني نهاية الحواف من هذا العنصر. بعد ذلك، وبتتبع الحقل NEXT للانتقال إلى العنصر التالي والذي يحوي البيانات (B) ، نكرر الطريقة السابقة ونتأكد من أن الشكل (6) هو تمثيل للمخطط الوارد في الشكل (5).

2.2.2 تمثيل المخططات باستخدام مصفوفات الجوار (Adjacency Matrices)

افرض أن لدينا المخطط الوارد سابقاً في الشكل (3) ونكرره هنا في الشكل (7) حيث يحتوي هذا المخطط على مجموعة العناصر {A, B, C, D, E} وعلى مجموعة الحواف {(A, B), (A,C), (A, D), (B, D), (C, D), (C, E), (D, E)}



شكل (7)

فيمكننا أن نمثل هذا المخطط باستخدام مصفوفة ذات بعدين، البعد الأول يتكون من خمسة صفوف، والبعد الثاني من خمسة أعمدة. أي نحتاج إلى مصفوفة أبعادها 5 * 5 لتمثيل هذا المخطط، بحيث أنه إذا كانت هناك حافة من العنصر (A) إلى العنصر (B) فإن الموقع داخل المصفوفة الذي ينتج عن الصف (A) والعمود (B) سوف يحتوي على القيمة «1».

وبشكل عام فإن مصفوفة الجوار (a i j) = A تعرف كما يلي:

$$a_{ij} = \begin{cases} 1 & (V_i, V_j) \text{ إذا كانت هناك حافة} \\ 0 & \text{إذا لم تكن هناك حافة} \end{cases}$$

وبصورة عامة، عزيزي الدارس، إذا كان عدد العناصر الموجودة في المخطط (n) فإن مصفوفة الجوار هي مصفوفة ذات بعدين وكل بعد يحتوي على عدد (n) من المواقع.

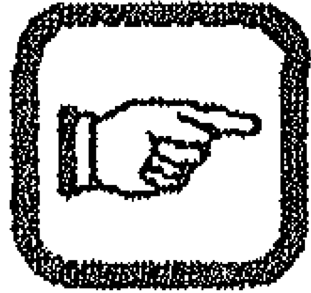
بالرجوع إلى الشكل (7) يمكن تمثيل هذا المخطط بمصفوفة الجوار التالية، (الشكل (8)):

	A	B	C	D	E
A	0	1	1	1	0
B	0	0	0	1	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	0	0

شكل (8)

نلاحظ أن مصفوفة الجوار تحتوي دائماً على العنصرين 0 أو 1 ولهذا السبب يطلق على مثل هذه المصفوفة المصفوفة البولينية (Boolean matrix) أو المصفوفة الثنائية (Bit matrix).

وتجدر الإشارة إلى أن مصفوفة الجوار A للمخطط G تعتمد على ترتيب عناصر المخطط. على سبيل المثال، يمكن إعادة ترتيب عناصر الشكل (7) للحصول على مصفوفة جوار مختلفة عن تلك الواردة في الشكل (8). والفارق الوحيد عند تغير ترتيب العناصر هو أننا نستبدل بعض الصفوف بصفوف أخرى، وكذلك بعض الأعمدة بأعمدة أخرى، كما هو موضح في المثال التالي.



مثال (3)

إذا استبدلنا A مكان C في مصفوفة الجوار للشكل (8) نحصل على مصفوفة الجوار المبينة في الشكل (9):

	C	B	A	D	E
C	0	0	0	1	1
B	0	0	0	1	0
A	1	1	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

شكل (9)

حيث يمكن الحصول على هذه المصفوفة من المصفوفة المبينة في الشكل (8) كما يلي:

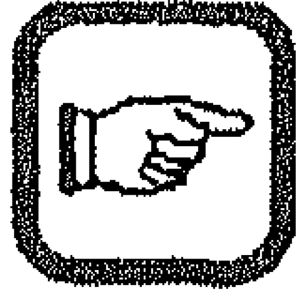
1. استبدل محتويات العمود الأول مع محتويات العمود الثالث.
2. بعد ذلك استبدل محتويات الصف الأول مع محتويات الصف الثالث لنحصل على مصفوفة الجوار التالية:

1	1	0	0	0		0	1	0	1	1
0	1	0	0	0		0	1	0	0	0
0	1	0	1	1	⇨	1	1	0	0	0
1	0	0	0	0		1	0	0	0	0
0	0	0	0	0		0	0	0	0	0

بعد استبدال العمود الأول بالعمود الثالث.

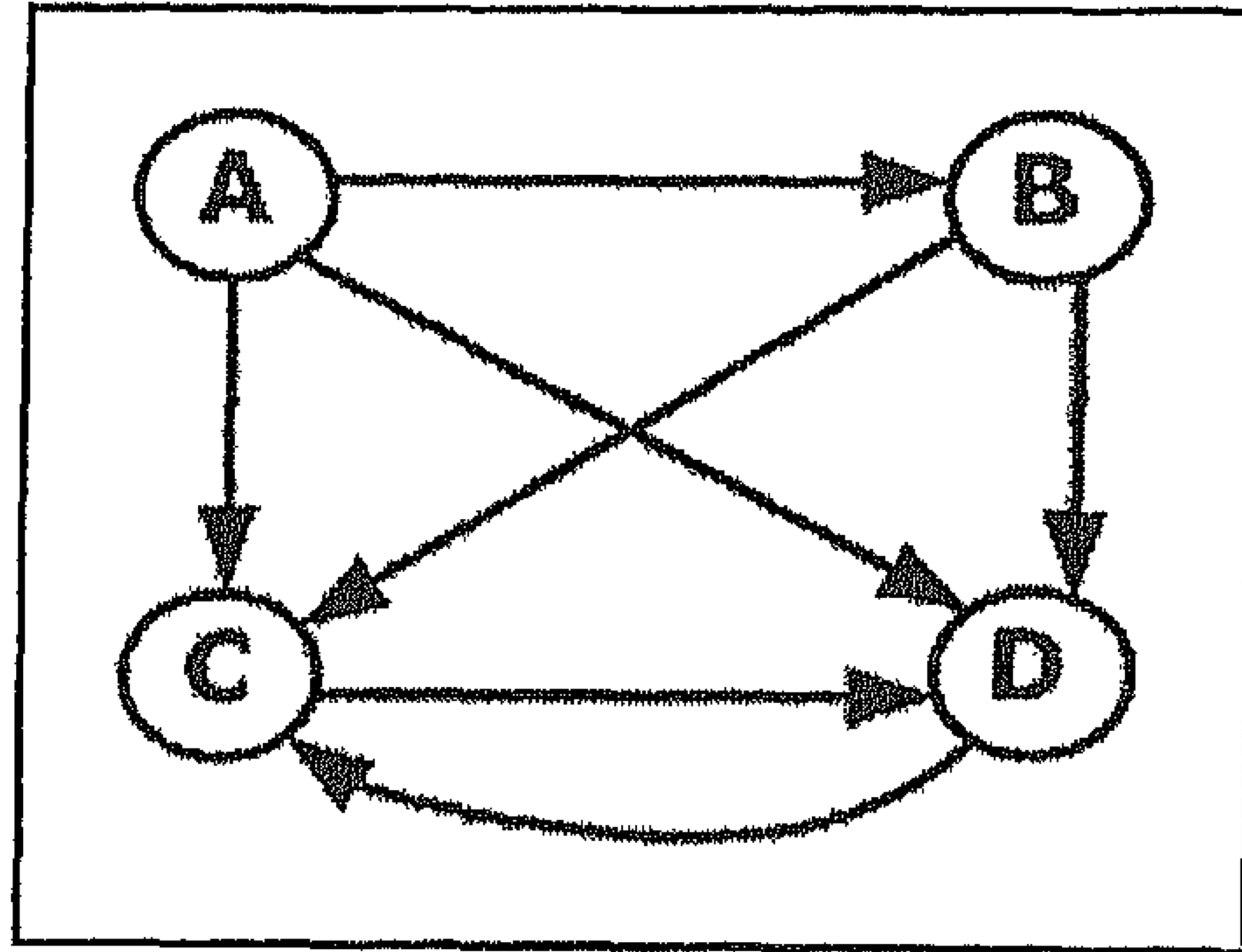
بعد استبدال العمود الأول مع العمود الثالث واستبدال الصف الأول مع الصف الثالث.

وفــي حالــة كــون المخطــط غيــر متجه فــإن مصفوفــة الجــوار ســوف تكــون متماثلة (Symmetric)، أي أن العنصر a_{ij} ســوف يكافئ العنصر a_{ji} في المصفوفة. والسبب في ذلك أن الحافــة (A, B) فــي المخطط غير المتجــه تكافئ الحافتين المتجهتيــن (A, B) و (B, A) وأن القيمــة المناظــرة للحافة (A, B) تماثلها القيمة 1 المناظــرة للحافة (B, A) في مصفوفة الجوار.



مثال (4)

افرض أن لديك المخطط المبين في الشكل (10) وأن ترتيب العناصر هو A, B, D, C أوجد مصفوفة الجوار.



الشكل (10)

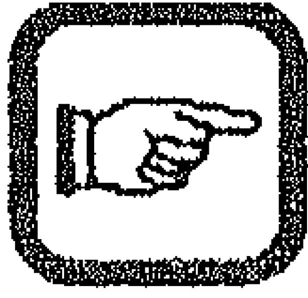
الحل:

$$Adjacency \quad Matrix = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

3. الشبكات

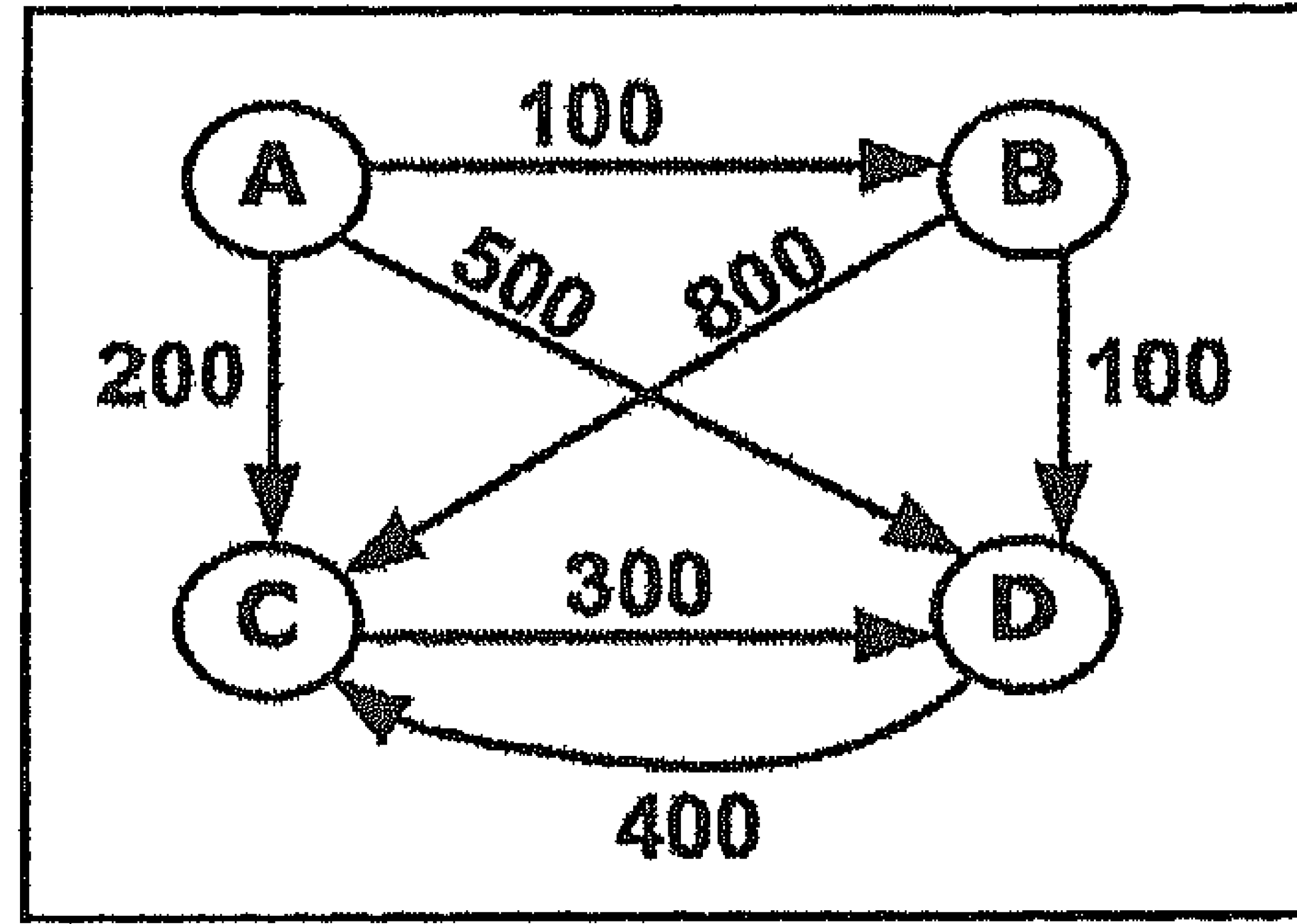
كما أسلفنا، عزيزي الدارس، يمكننا أن نضيف وزنا معيناً أو تكلفة معينة مع كل حافة من حواف المخطط. فعلى سبيل المثال إذا فرضنا أن المخطط يمثل مدن فلسطين وأن الحواف تمثل طول الطرق بين المدن المختلفة، أو كلفة التنقل بين المدن، يمكننا وضع أطوال هذه الطرق أو كلفة التنقل على الحواف. ويمكننا الحصول على نوع آخر من مصفوفة الجوار تحتوي على الأوزان والتكلفة بدلاً من القيم 1 و0.

وعندئذ يسمى المخطط شبكة (Network). وعليه فإن الشبكة تعرَّف على أنها مخطط لكل حافة من حوافه قيمة عددية موجبة تمثل وزن تلك الحافة.



مثال (5)

افرض أننا أضفنا إلى الشكل (10) السابق أوزاناً كما هو مبين في الشكل(11) التالي:



الشكل (11)

فيمكن تمثيل هذا المخطط بمصفوفة أوزان الجوار (Cost Adjacency matrix) التالية على فرض أن ترتيب العناصر هو (A, B, C, D):

	A	B	C	D
A	0	100	200	500
B	0	0	800	100
C	0	0	0	300
D	0	0	400	0

مصفوفة أوزان الجوار

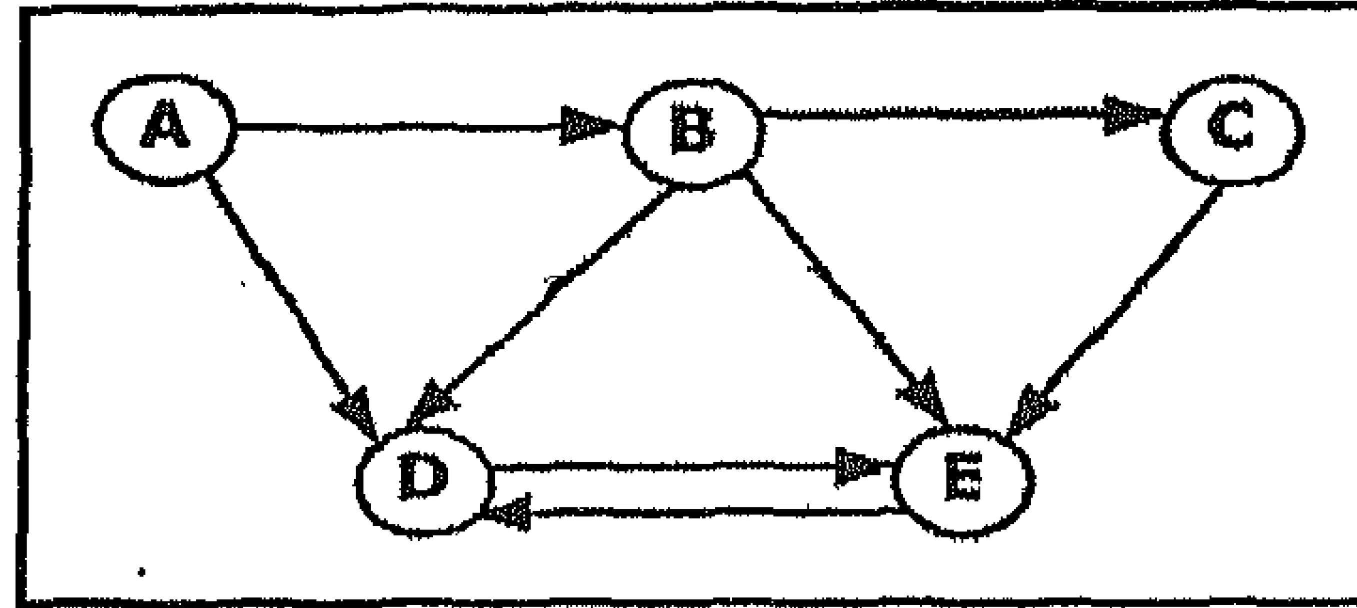
يمكننا استخدام مثل هذه المصفوفة لإيجاد أقصر الممرات عند الرغبة في الوصول من مكان إلى آخر في المخطط، وهذا هو موضوع البند التالي.



أسئلة التقويم الذاتي (1)

بينّ كيف يُمَثَّلُ المخطط المبين في الشكل (12) باستخدام مؤشرات الربط حيث أن قوائم الاتصال كما يلي:

العنصر	قائمة الجوار
A	B, C
B	C, D, E
C	E
D	E
E	D



شكل (12)

إيجاد أقصر الممرات في الشبكات

سنحاول، عزيزي الدارس، في هذا البند من الوحدة التوصل إلى خوارزمية لإيجاد أقصر الممرات بين نقطتين في مخطط معين تحمل حوافه أوزاناً معينة. وكما أشرنا في السابق يمكن للمخطط أن يمثل شبكة الطرق المحلية أو شبكة الطرق الجوية العالمية. حيث تمثل رؤوس المخطط المدن، وتمثل حوافه الطرق التي تربط المدن معاً. ومن الحكمة أن يسأل سائق المركبة نفسه عن أقصر الطرق للسفر من مدينة إلى أخرى إذا كانت هناك طريق تؤدي به للوصول إلى وجهته، حيث أن الإجابة على مثل هذا السؤال تؤدي إلى توفير الوقت والجهد والتكلفة. وسنقتصر شرحنا على المخططات المتجهة وذلك لأخذ الطرق باتجاه واحد بعين الاعتبار.

يمكننا كتابة المسألة التي نحن بصدد كتابة خوارزمية لحلها كما يلي:

إذا أعطينا مخططاً متجهاً G، بحيث أن كل حافةٍ من حوافه لها وزن أو تكلفة معينة ممثلة بعدد صحيح، فما هي أقصر الممرات المؤدية من نقطة المصدر (1) إلى جميع النقاط المتبقية في المخطط؟

وعلى سبيل المثال افرض أن لدينا المخطط المتجه كما في الشكل (13)، والأوزان المبينة. وافرض أيضاً أن النقطة (1) هي نقطة الانطلاق، فيمكننا الاستنتاج على سبيل المثال بأن أقصر الممرات من النقطة (1) إلى النقطة (2) هي عن طريق (1←3←4←2) وأن طول هذا الممر هو15 + 20 + 25 = 60 مع العلم أن هناك طريقاً مباشرة من 1 إلى 2 ولكن طول هذا الطريق هو 65.

الشكل (13)

ولحل المسألة بصورتها العامة، أي إيجاد خوارزمية لتحديد أقصر الممرات بين نقطتين في مخطط معين تحمل حوافه أوزاناً معينة، نبدأ من مصفوفة تكلفة الجوار (Cost Adjacency Matrix)، وتجدر الإشارة إلى أن أول من أوجد مثل هذه الخوارزمية هو العالم Dijkstra ديجكسترا.

فلو فرضنا ان المخطط يتكون من العناصر 1، 2، ...، N فإن هذه الخوارزمية تحتاج إلى مصفوفة تكلفة الجوار والتي سنرمز إليها بالاسم Cost وإلى ثلاثة مصفوفات أخرى كما يلي:

1. المصفوفة DISTANCE لتخزين طول أقصر الطرق من نقطة الانطلاق إلى النقطة j وذلك في الموقع DISTANCE [j]. ولذلك فإن هذه المصفوفة هي ذات بعد واحد.
2. المصفوفة PATH وذلك لتخزين العناصر التي نمر بها في أقصر الممرات حيث أن PATH [j] يحتوي على النقطة السابقة المباشرة للنقطة j في أقصر الممرات.

3. المصفوفة INCLUDED وهي مصفوفة بولينية منطقية تحتوي على العناصر المشمولة، بمـا في ذلك نقطة الأصـل (الانطلاق) ضمن أقصر الممرات لغايـة هذه اللحظة من تطبيق الخوارزميـة بدءاً من نقطة الانطلاق. ففي حالة شـمول أي مـن النقاط قبل J ضمن أقصر الممرات تصبح قيمة Included [j] تساوي true .

والمثـال رقـم (6) الوارد لاحقاً، يبين قيـم هذه المصفوفات في المراحـل المختلفة لتنفيذ خوارزمية إيجاد أقصر الممرات على مصفوفة تكلفة الجوار الواردة هناك.

وبينمـا تتغيـر قيم DISTANCE [j] وPATH [j] لأي من النقـاط j والواقعة على أقصر الممرات طالما أن البرنامج شغّال يتوقف تغير هذه القيم حين يتم تعديل قيمة INCLUDED[j] من false إلى true لأن هذا يعني أن DISTANCE [j] أصبح يحتوي على أقصر المسافات من نقطة الانطلاق إلى النقطة j.

ويمكننا الإعلان عن هذه المصفوفات بالإضافة إلى مصفوفة تكلفة الجوار بلغة سي/ سي++ على النحو التالي على فرض أن المخطط يحتوي على العناصر من 0 إلى N-1 .

```
int  COST[N][N];
int  DISTANCE[N];
int  PATH[N];
bool  INCLUDED[N];
```

وتبيـن خوارزمية Dijkstra (ديجكســترا) لإيجاد أقصر الممرات ابتـداءً من نقطة معينة تسـمى نقطة المصدر (source) وانتهاء بنقطة أخرى تسمى نقطة الوجهة (Destination). وتحتوي هذه الخوارزمية على الدوال التالية:

1. الدالـة ReadMAT: حيـث يتم عن طريق هـذه الدالة قراءة مصفوفـة أوزان الجوار. وعند إدخال مثل هذه القيم يجب إدخال القيمة 0 (صفر) في حالتين هما:

   أولاً: إذا كانت الحافه (Vi, Vi) أي من والى نفس النقطة.

   ثانياً: إذا لم توجد حافة.

2. دالة Initialize حيث يتم في هذه الدالة تحديد القيم الابتدائية التالية:

   أ - INCLUDED [SOURCE] تحدد في البداية على انها تساوي true.

   ب- INCLUDED [i] لكل النقاط المتبقية تساوي false.

   ج- يتم تحديد القيم الابتدائية للمصفوفة DISTANCE كما يلي:

   إذا كانت I هي المصدر 0

   إذا كان M [source, i]!= 0 DISTANCE [i] =M [Source, i]

إذا لم تكن هناك حافة بين المصدر والنقطة i MAXINT

والمقصـود بالقيمـة MAXINT هو أكبر قيمة عددية صحيحـة (maximum int) يمكن تمثيلها في الحاسوب وذلك عندما تكون الأوزان من النوع الصحيح. وهذه القيمة تعتمد على نوع الحاسوب.

أما إذا كانت الأوزان قيماً حقيقية فيمكن تمثيل هذه القيمة بأكبر عدد حقيقي.

أما إذا نظرنا إلى القيمة MAXINT بدون الرجوع إلى الحاسوب فيمكن تمثيل هذه القيمة بما لا نهاية ∞ .

د- يتم تحديد القيم الابتدائية في المصفوفة PATH كما يلي:

إذا كان M [Source, i]!= 0 SOURCE=PATH[i]

في الحالات الأخرى MAXINT

3. الدالـة FINDMIN: حيـث أن FINDMIN[j] تعيـد أقصر الممرات من نقطـة البداية إلى النقطـة j . وبذلـك تصبح النقطة j ضمن أقصر الممرات، حيث يتم تعديل قيمة Included [j]لتصبح مسـاوية للقيمة true. ولتحديد ذلك يتم فحص أبعاد جميع الممرات من نقطة البداية إلى النقطة j، وهذه الأبعاد تخزن في المصفوفة Distance.

4. الدالة UPDATE: تستخدم لتعديل المصفوفة DISTANCE والمصفوفة PATH في حالة إيجاد نقطة مثل i حيث أن (INCLUDED [i] = false) تعطي ممر طوله أقصر الممرات الموجودة قبل اعتبار هذه النقطة وفي هذه الحالة تضاف هذه النقطة إلى النقاط المشمولة في أقصر الممرات.

5. الدالة all: وتستخدم كدالة منطقية وتعيد القيمة true عندما تصبح قيم جميع المواقع في المصفوفة INCLUDED تساوي true، وتعيد هذه الدالة القيمة false خلاف ذلك.

6. الدالة ShortPath وتسـتخدم كدالة يتم من داخلها اسـتدعاء الدوال السابقة بالإضافة إلى دالـة للطباعة التي تقوم بطباعة النتائج وأي معلومات أخـرى نرغب فيها. وفي نهاية هذه الدالة يتم طباعة أقصر الممرات وتكلفة هذا الممر.

7. دالة الطباعة Print: تسـتخدم هذه الدالة لطباعة قيم المصفوفات DISTANCE، PATH، INCLUDED بعد كل خطوة من خطوات الخوارزمية.

8. الدالة PrintPath: يسـتخدم هذا الإجراء لطباعة أقصر الممرات بين نقطة المصدر ونقطة الوجهة المقصودة.

9. لقد تم تحويل هذه الخوارزمية إلى برنامج بلغة سي/سـي++، كما تم تنفيذ البرنامج على المثال (6) والتدريب (1) التاليين وحصلنا على النتائج المبينة هناك.

ولزيادة الاستفادة يمكنك،عزيزي الـدارس، كتابة الدوال الواردة في الخوارزمية للحصول على برنامج سي++ كامل تستطيع استخدامه في مسائل متعددة. وفيما يلي وصف لخوارزمية Dijkstra لإيجاد أقصر الممرات:

```
const int N=50;//Assuming that the graph with <= 50 nodes
typedef bool BooleanType[N];
int  COST[N][N];
int  DISTANCE[N];
int  PATH[N];
bool  INCLUDED[N];
//////*********************************************/////

void SHORTPATH (int SOURCE);
{/*This function Determines the shortest path from source to all
   other  nodes in the graph represented by the adjacency matrix cost.
   The global array DISTANCE stores shortest distances, and the global
   array PATH stores the nodes on the shortest path*/
 BooleanType Included;
 int i,j;
   ReadMat();  //Initialize  The  Arrays  COST,  DISTANCE,  PATH  and
INCLUDED
 Initialize(SOURCE);
 do
  {FINDMIN(j);
  INCLUDED[j]=true;
  for(i=0;i<N;i++)
   if (!INCLUDED[i])
       UPDATE(i,j); //Trying to change Distance[i] and PATH[i]
                 //depending on theinclusion of Node j
  Print;
  }
 while ALL(Included);
 PrintPath;
}
```

مثال (6)

افرض أن لدينا مصفوفة تكلفة الجوار التالية:

	1	2	3	4	5
1	0	800	2985	310	200
2	800	0	410	612	0
3	2985	410	0	1421	0
4	310	612	1421	0	400
5	200	0	0	400	0

حيث أن تكلفة النقطة إلى نفسها تساوي 0 (صفراً). وكذلك إذا لم تكن هناك حافة من النقطة i إلى النقطة j فإن قيمة الموقع في الصف i والعمود j تساوي ∞. وعند بدء البرنامج، فإن الدالة المسماة Initialize تعمل على تحويل الأصفار الناتجة عن عدم وجود حواف بين النقاط في المخطط إلى أكبر قيمة صحيحة ممكنة في لغة سي++ ويرمز إلى هذه القيمة (أو الثابت الصحيح) بالرمز MAXINT وهو يساوي32767، عند استخدام الحواسيب المصغرة وهذه القيمة هي 1 - 215 بسبب تخزين الأعداد الصحيحة في النظام المكمل لاثنين وذلك باستخدام ستة عشر موقعاً ثنائياً.

أمّا من ناحية نظرية فيمكن اعتبار أن تكلفة الجوار بين نقطتين لا توجد بينهما مسافة على أنها مساوية للقيمة ما لا نهاية. وبما أن الحاسوب لا يمكن له أن يفهم القيمة ما لا نهاية (∞) لذلك اعتبرنا أن تكلفة الجوار في مثل هذه الحالة تساوي أكبر قيمة صحيحة يمكن تمثيلها في الحاسوب.

وهذه القيمة معرفة في لغة سي/سي++، وهي التي استخدمناها. فعند تنفيذ البرنامج على مصفوفة تكلفة الجوار المعطاة على فرض أننا نرغب في إيجاد أقصر الممرات من النقطة 1 إلى النقطة 3، نحصل على ما يلي:

```
Initial states of the matrices          القيم الابتدائية للمصفوفات
DISTANCE:        0      800    2985   310    200
PATH:            0      1      1      1      1
INCLUDED:        true   false  false  false  false
          بعد ذلك نحصل على القيم التالية لهذه المصفوفات
DISTANCE:        0      800    2985   310    200
PATH:            0      1      1      1      1
INCLUDED:        true   false  false  false  true
```

DISTANCE:	200	1000	1821	400	0
PATH:	5	1	4	5	0
INCLUDED:	true	false	false	true	true

DISTANCE:	200	1000	1410	400	0
PATH:	5	1	2	5	0
INCLUDED:	true	true	false	true	true

DISTANCE:	200	1000	1410	400	0
PATH:	5	1	2	5	0
INCLUDED:	true	true	true	true	true

The Path is 1 → 2 → 3

تبدأ خوارزمية ديجكسترا بقراءة قيم مصفوفة الجوار، ويتم تخزين تلك القيم في المصفوفة COST . وتكون تلك القيم كما هو مبين في المثال (6) . فعلى سبيل المثال تُساوي COST [2,3] القيمة 410 . وهذا يعني أن وزن الحافة التي تصل بين النقطتين 2 و3 هو 410 .

بعد ذلك يتم استدعاء الإجراء Initialize، والذي يُحدّد القيم الابتدائية للمصفوفات DISTANCE وPATH وINCLUDED كما يلي:

- محتويات المصفوفة DISTANCE الابتدائيّة:

بُعْدُ النقطة 1 من نقطة البداية (وهي النقطة 1 نفسها)، أي [1] DISTANCE يُساوي 0،

بُعْدُ النقطة 2 من نقطة البداية، أي [2] DISTANCE يُساوي 800،

بُعْدُ النقطة 3 من نقطة البداية، أي [3] DISTANCEيُساوي 2985،

بُعْدُ النقطة 4 من نقطة البداية، أي [4] DISTANCE يُساوي 310،

وبُعْدُ النقطة 5 من نقطة البداية، أي [5] DISTANCE يُساوي 200،

وبذلك تكون المصفوفة DISTANCE كما يلي:

DISTANCE: 0 800 2985 310 200

- أما بالنسبة للقيم الابتدائية في المصفوفة PATH، فهي قيمة نقطة البداية في حالة وجود حافة أو ممرّ من نقطة البداية إلى النقاط الأخرى. ولهذا تكون محتويات المصفوفة PATH مساوية للقيمة 1 (أي نقطة البداية) باستثناء [1] PATH، حيث تساوي قيمته صفراً.

- وأما بالنسبة للمصفوفة INCLUDED، فإن قيم جميع عناصرها تساوي false باستثناء قيمة العنصر [1] INCLUDED (حيث أن القيمة 1 تمثل نقطة البداية) والتي تساوي true. وهذا يعني أن النقطة 1 ينبغي أن تُشمل في أقصر الممرّات؛ فهذا منطقيّ ولا بدّ منه.

وبذلـك نحصل على القيم الابتدائية للمصفوفـات DISTANCE وPATH وINCLUDED كما يلي:

DISTANCE:	0	800	2985	310	200
PATH :	0	1	1	1	1
INCLUDED:	true	false	false	false	false

وفـي المرحلة التالية من تنفيذ الخوارزمية يتم تعديل محتويات المصفوفات الثلاث بحيث تعكـس تلك التعديلات أبعاد أقصر الممرّات من نقطة البداية. كما تبيّن هذه التعديلات النقاط التي تمّ أخذها بعين الاعتبار والنقاط المشمولة في أقصر الممرّات.

لذلـك يتم، في الخطوة الثانية مـن الخوارزميّة، إجراء التعديـلات التالية على المصفوفات الثـلاث: فبما أن بُعْـدَ النقطة 5 عن النقطة 1 هو الأقل من أبعاد النقطة الأخرى، فإن النقطة 5 ينبغي أن تكون ضمن أقصر الممرّات. وبذلك تصبح [5] INCLUDED مساوية للقيمة true . وتبقـى النقطـة 1 بوصفها النقطة التي يمكن المرور بها، حتى هذه اللحظة من الخوارزميّة، انطلاقـاً من نقطة البداية (وهي النقطة 1) مباشـرةً إلى النقـاط الأخرى. كما تبقى محتويات المصفوفة PATH كما هي. وعليه تكون محتويات المصفوفات الثلاث كما يلي:

النقطة	1	2	3	4	5
DISTANCE:	0	800	2985	310	200
PATH :	0	1	1	1	1
INCLUDED:	true	false	false	false	false

دعنا، عزيزي الدارس، نوضّح معنى القيم الموجودة في المصفوفات الثلاث، ولنأخذ الموقع أو النقطة 2 على سبيل المثال:

نلاحـظ أن محتويـات الموقع الثاني فـي المصفوفات الثلاث هـي (false, 1, 800). وهذا يعني أن طول أقصر الممرّات من النقطة 1 إلى النقطة - 2 حتى هذه اللحظة - هو 800 . كما تعنـي القيمة 1 أن النقطة التي تم اسـتخدامها للوصول إلى النقطـة 2 هي النقطة 1. وتعني القيمة false أن النقطة 2 لم تؤخذ بعين الاعتبار لتحديد أقصر الممرّات حتى هذه اللحظة.

وفي الخطوة التالية نلاحظ ما يلي:

يمكن الوصول إلى النقطة 3 إما من النقطة 1 مباشرة - وتكلفة هذا الممرّ 2985 - أو عن طريق النقطة 2 - وتكلفة هذا الممرّ 800 - من النقطة 1، أو عن طريق النقطة 4، وتكلفة الوصول في هذه الحالة هي 310. وبما أن 310 أصغر من 800، لذلك يتم شمول النقطة 4 ضمن أقصر الممرّات. وعليه، تصبح قيمة DISTANCE [3] مساوية لبُعْدِ النقطة 3 عن النقطة 4 مضافاً إليه بُعْدُ النقطة 4 عن النقطة 1، وهو يساوي (310 + 1421 = 1731). كما تصبح محتويات 4 = [3] PATH، أي أن الوصول إلى النقطة 3 عن طريق النقطة 4.

وبما أن 1 = [4] PATH، أي أن الوصول إلى 4 عن طريق النقطة 1 (نقطة البداية)، وتمّ اختيار النقطة 4 للوصول إلى النقطة 3، تصبح محتويات العنصر الرابع للمصفوفة INCLUDED مساوية للقيمة true، أي أن محتويات المصفوفات الثلاث تصبح كما يلي:

النقطة	1	2	3	4	5
DISTANCE:	0	800	1731	310	200
PATH :	0	1	4	1	1
INCLUDED:	true	false	false	true	true

وطالما أن بعض قيم المصفوفة INCLUDED غير مأخوذة بعين الاعتبار (وهو ما تشير إليه القيمة false)، يستمرّ تنفيذ الخوارزمية حتى تصبح جميع قيم عناصر تلك المصفوفة مساوية للقيمة true.

إذن، في الخطوة التالية من الخوارزميّة، سنلحظ أن طول الممرّ من 1 إلى 2 ثمّ إلى 3 أصغر من طول الممر من 1 إلى 4 ثم إلى 3 على الرغم من أن طول الممر من 1 إلى 2 أكبر من طول الممر من 1 إلى 4 . ونتيجة لذلك يصبح طول المسافة من 1 إلى 3 مساوياً للقيمة (800 + 410 = 1210)، وهو أقل من الطول السابق للممر من 1 إلى 4 ثم إلى 3.

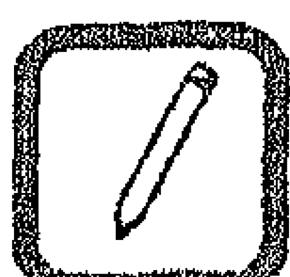
ولذلك يتم إجراء التعديلات اللازمة على المصفوفات عن طريق الإجراء UPDATE. ومن ضمن تلك التعديلات، في هذه الخطوة، تعديل يغيّر قيمة PATH[3] من 4 إلى 2، وتعديل DISTANCE[3] من 1731 إلى 1210، وتعديل قيمة INCLUDED[2] من false إلى true، مما يعني أن النقطة 2 قد أخذت بعين الاعتبار. وعليه تصبح محتويات المصفوفات الثلاث كما يلي:

النقطة	1	2	3	4	5
DISTANCE:	0	800	1210	310	200
PATH :	0	1	2	1	1
INCLUDED:	true	true	false	true	true

وبمـا أن محتويـات [3] INCLUDED مـا زالـت مسـاوية للقيمة false، يسـتمر تنفيذ الخوارزميّـة. وبمـا أن النقطة 3 هي نقطة النهاية في الوقت الذي تـمّ فيه أخذ النقاط الأخرى جميعها بعين الاعتبار، والطول الحالي للممر من 1 إلى 3 هو أقصر الممرّات الأخرى من جميع النقاط، تصبح قيمـة [3] INCLUDED مسـاويـة للقيمـة true . وبذلك يتوقّف الاسـتدعاء المستمرّ للإجراء UPDATE. وعند هذه اللحظة تكون محتويات المصفوفات الثلاث كما يلي:

النقطة	1	2	3	4	5
DISTANCE:	0	800	1210	310	200
PATH :	0	1	2	1	1
INCLUDED:	true	true	true	true	true

ومن المصفوفة الأخيرة نجد أقصر الممرّات وذلك بالنظر إلى النقطة 3 (أي نقطة النهاية)، حيث نلاحظ أن 2 = [3] PATH، أي أن الوصول إلى 3 تمّ من 2. ويرمز لهذا الممرّ بالرمز (2 ← 3). أما عن كيفية الوصول إلى 2، فمن الواضح أنه ـ تم عبر 1 لأن 1 = [2] PATH . وبما أن 1 هي نقطة البداية فإن أقصر الممرّات هو 1 ← 2 ← 3 . ولمعرفة طول هذا الممر نستخدم المصفوفة DISTANCE، فتعطينا [3] DISTANCE القيمة (1210) وهي طول هذا الممر.



تدريب (1)

أوجد أقصر الممرات إذا أردت المسير من النقطة 5 إلى النقطة 3 في المخطط الممثل بمصفوفة تكلفة الجوار في المثال (6) السابق.

يمكن تعميم مسـألة اقصر الممرات لتشـمل ايجاد اقصر الممرات مـا بين كل زوجين من العناصر v_i وv_j، حيث أن i لا تساوي j.

ويمكن حل هذه المسألة بإحدى طريقتين:

أولاً: عن طريق تطبيق خوارزمية Dijkstra مرة لكل عنصر من العناصر باعتبار هذا العنصر علـى أنه المصدر (نقطة الانطلاق) وبذلك فإننا نحتاج إلى تنفيذ خوارزمية Dijkstra n من المرات حيث أن n تساوي عدد العناصر الموجودة في المخطط.

ثانياً: باستـخدام خوارزميـة أخـرى أُطلـق عليهـا خوارزميـة فلويـد .(R.W. Floyd) وهذه الخوارزمية هي تطوير على خوارزمية Dijkstra السـابقة. وباستخدام هذه الخوارزمية يتم حساب مصفوفة أقصر الممرات ما بين أي نقطتين مختلفتين في المخطط.

وفيمـا يلي، عزيزي الدارس، وصـف للدالة التي تمثل طريقة فلويد في حسـاب مصفوفة أقصر الممرات:

```
VOID FLOYDMINCOST(MATRIX COST,MATRIX& PATH,MATRIX P)
 {INT   I, J, K;
  //PATH IS TH RESULTANT MATRIX AND CONTAINS THE VALUES
  //INCLUDED IN THE SHORTEST PATH BETWEEN NODES I AND J
  //AS SEEN IN PATH [I,J]
  FOR(I=0 ;I<N;I++)
   FOR(J=0 ;J<N;J++)
    {P[I][J]=COST[I][J];
     PATH[I][J]=0;
     }
  FOR(I=0 ;I<N;I++)
   P[I][J]=0;
  FOR(K=0 ;K<N;K++)
   {FOR(I=0 ;I<N;I++)
     {FOR(J=0 ;J<N;J++)
       {IF (P[I][K]+P[K][J]<P[I][J])
         {P[I][J]=P[I][K]+P[K][J];
          PATH[I][J]=K;
          }
        }
      }
    }
}
```

وتجدر الإشارة إلى أن هذه الدالة قد تستخدم الإعلان العام التالي:

```
const int N=100;
typedef int   Matrix[N][N];
```

تستخدم هذه الخوارزمية ثلاثة مصفوفات هي:

1. مصفوفة تكلفة الجوارCOST،

2. المصفوفـة PATH وتحتوي هـذه المصفوفة على القيم العددية لتكلفة الممر والمشـمولة بأقصر الممرات ما بين نقطتين i وj ويشار إلى ذلك بـ PATH [i][j].

3. المصفوفة P حيث يتم تهيئة هذه المصفوفة في بداية الخوارزمية كما يلي:

إذا لم تكن COST [i][j] مساوية للصفر COST [i][j]=P[i][j]

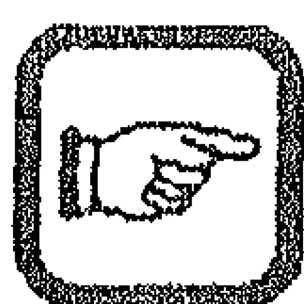
إذا كانت j=i (أي عناصر القطر) تساوي 0

في الحالات الأخرى MAXINT أو ∞

ونجد من الخوارزمية، وبالأخص من الدورة الداخلية أنه بعد الدورة k نحصل على:

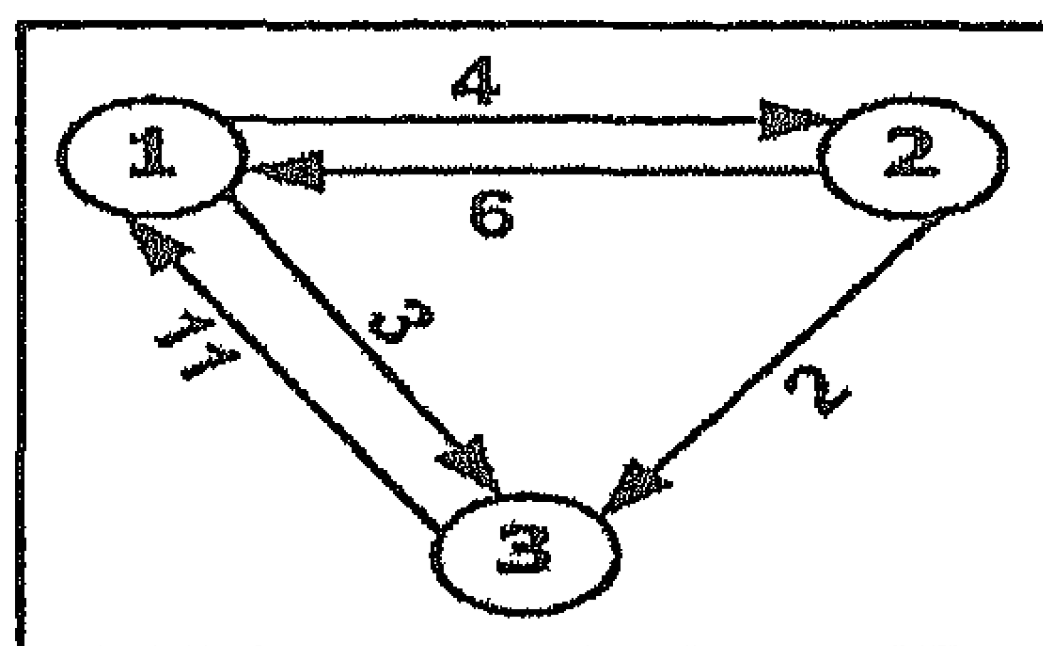
$$P [i][j] = MINIMUM (P [i][k] + P [k][j], P [i][j])$$

وهـذا بدوره يضمن لنـا أقصر الممرات من النقطة i إلى النقطـة j دون المرور خلال أي نقطة رقمها أكبر من j على فرض أن نقاط المخطط مرقمة 0 إلى N-1.



مثال (7)

أوجد مصفوفة أقصر الممرات للمخطط المتجه المبين في الشكل (14) التالي:



الشكل (14)

الحل:

أولاً نجد مصفوفة تكلفة الجوار كما يلي:

	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

بمـا أن عدد نقاط المخطط يسـاوي 3 نحصل على المصفوفات التاليـة بعد تنفيذ الإجراء FloydMinCost:

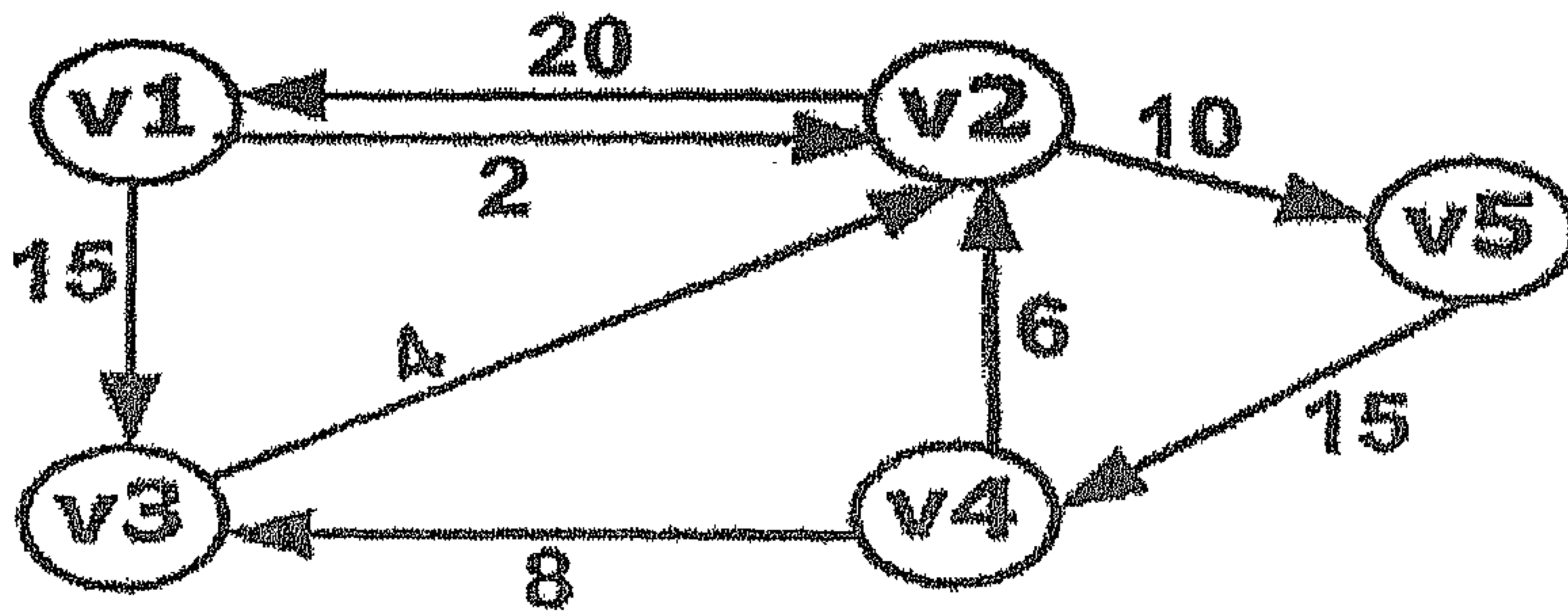
$$P^1 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 11 \\ 2 & 6 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array} \qquad P^2 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 6 \\ 2 & 6 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array} \qquad P^3 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 6 \\ 2 & 5 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array}$$

حيث أن P3 تمثل مصفوفة أقصر الممرات.



أسئلة التقويم الذاتي (2)

استخدم خوارزمية أقصر الممرات للحصول على طول أقصر الممرات مرتبة ترتيباً غير تنازلـي (Nondecreasing) بـدءاً من النقطة V1 إلى جميـع النقاط الأخرى في المخطط المبين في الشكل (15) التالي:



شكل (15)

4. العلاقة بين الهياكل الشجرية والمخططات

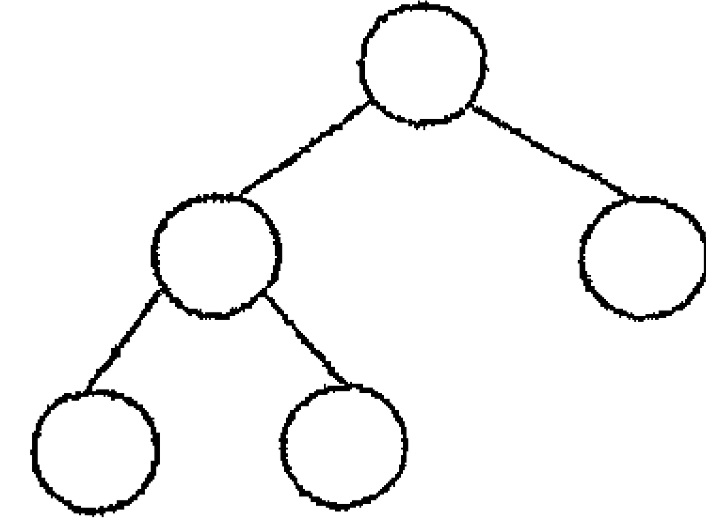
تجدر الإشـارة، عزيزي الدارس، إلى أنه يمكن اعتبار جميع الهياكل الشجرية، سواء أكانت ثنائيـة أم عامة، مخططات. لكن العكس غير صحيح، وذلك لوجود بعض القيود على الهياكل الشجرية، وغير الموجودة على المخططات.

ومن تلك القيود أن الشجيرات الفرعية لعنصر ما يجب أن لا تحتوي على عناصر مشتركة، أي أن عناصـر شـجيرة فرعية ما يجب أن تكون مختلفة تماماً عن عناصر أي شـجيرة فرعية أخرى لذلك العنصر، وهذا القيد غير موجود في حالة المخططات.

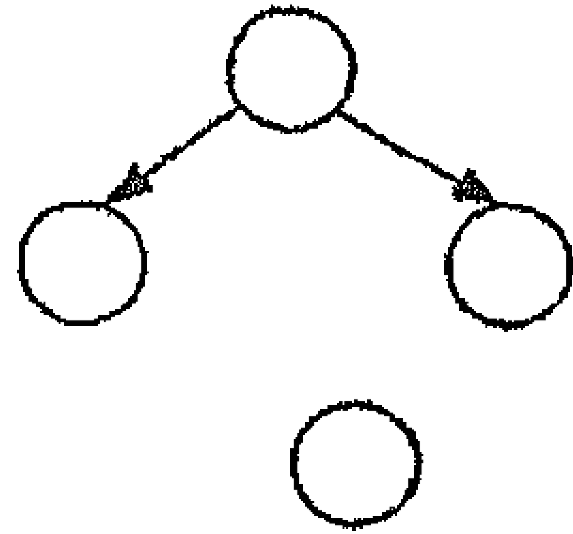
كذلك فإن الشيء الدارج في الهياكل الشجرية، أن مؤشرات الربط، تتجه من الأب إلى الابن وليس العكس (باستثناء بعض الحالات الخاصة). ومثل هذا القيد غير موجود في المخططات، إذ يمكـن لأي عنصر في المخطط أن يشـير إلى أي عنصر آخـر. وبمعنى آخر فإن علاقة الأب والابـن الموجـودة في حالة الهياكل الشـجرية، غير موجودة في حالـة المخططات. فالهياكل الشجرية هي عبارة عن مخططات متجهة لا تحتوي على دورات (cycles).

وبعبارةٍ أخرى فإن الهيكل الشجري هو حالة خاصة من المخططات (الشكل (16)) يتميز بأنه:

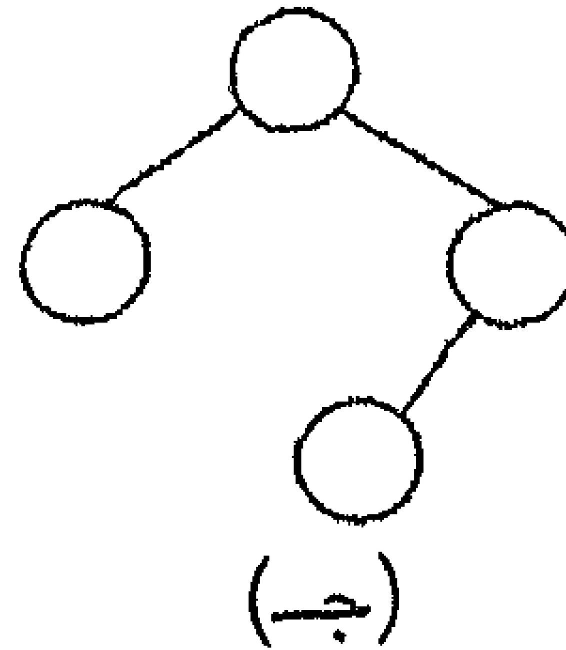
1. متصل (connected) (عد إلى تعريف المخطط المتصل).
2. ولا توجد به أي دورات (cycles)



(أ) مخطط متصل وبدون دورات



(ب) هذا مخطط غير متصل



(جـ)

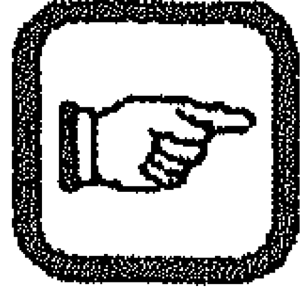
هذا الشـكل متصل ولا يوجد فيه دوران حيث أنه يمكن من أي نقطة زيارة النقاط الأخرى على الرغم من عدم وجود دوران

الشكل (16)

ويقال للمخطط بأنه متَّصِل إذا كان هناك ممر بين أي نقطتين من نقاطه.

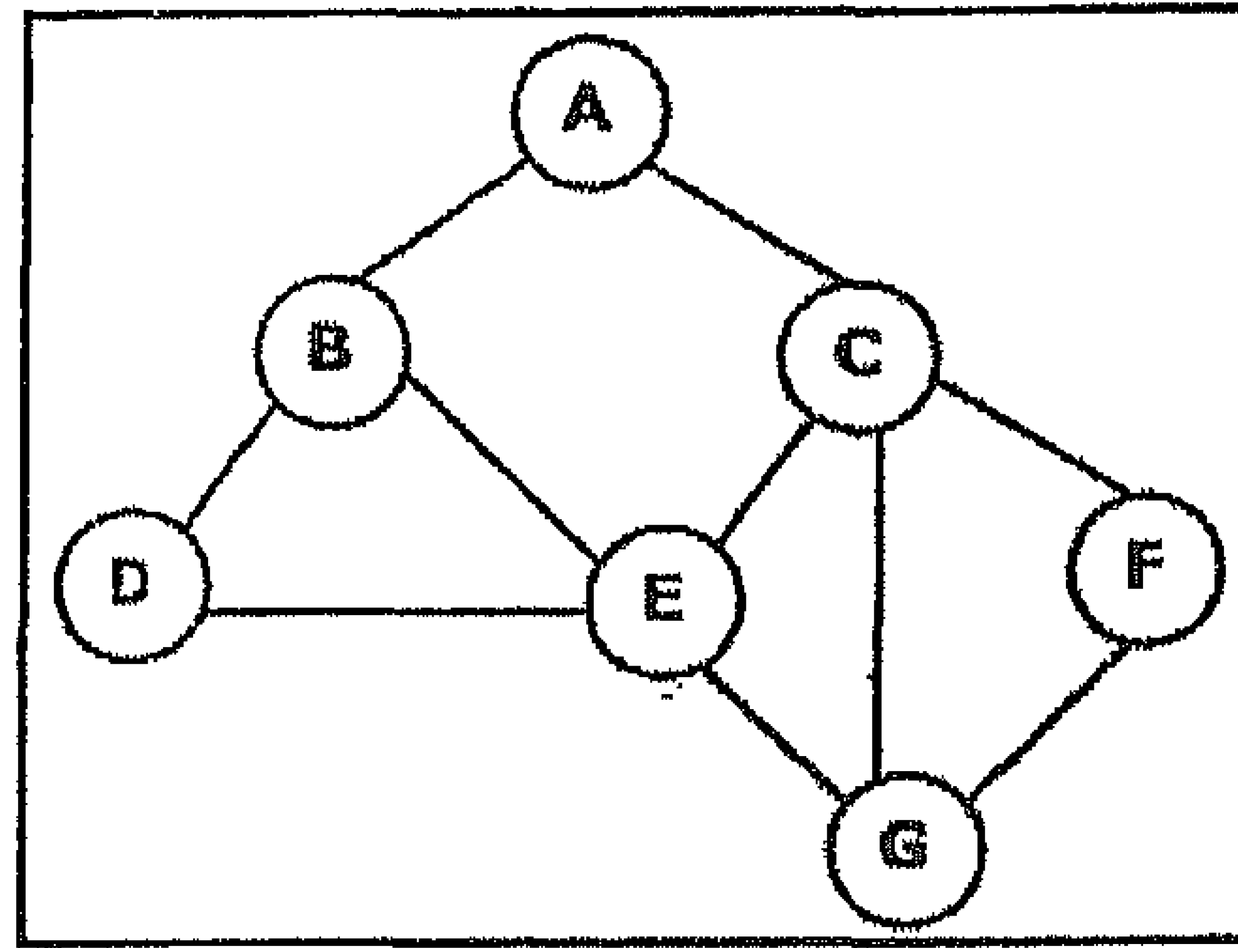
وتجدر الإشارة إلى أنه إذا كان عدد العناصر الموجودة في المخطط المتَّصِل يُساوي n وعدد الحواف يُساوي (n-1)، فلا يمكن لهذا المخطط أن يحتوي على أي دورات (cycles). ولهذا فإن مثل هذا المخطط يمثل هيكلاً شجرياً.

وكذلك فإنه من الممكن تحويل أي مخطط مُتَّصِلٍ إلى هيكل شجري بإزالة الحواف التي تعمل دورات حتى يتم الحصول على هيكل شجري. ويطلق على مثل هذا الهيكل الشجري الذي يتم الحصول عليه اسم الهيكل الشجري المتسع أو الممتد (Spanning Tree) للمخطط الأصلي.



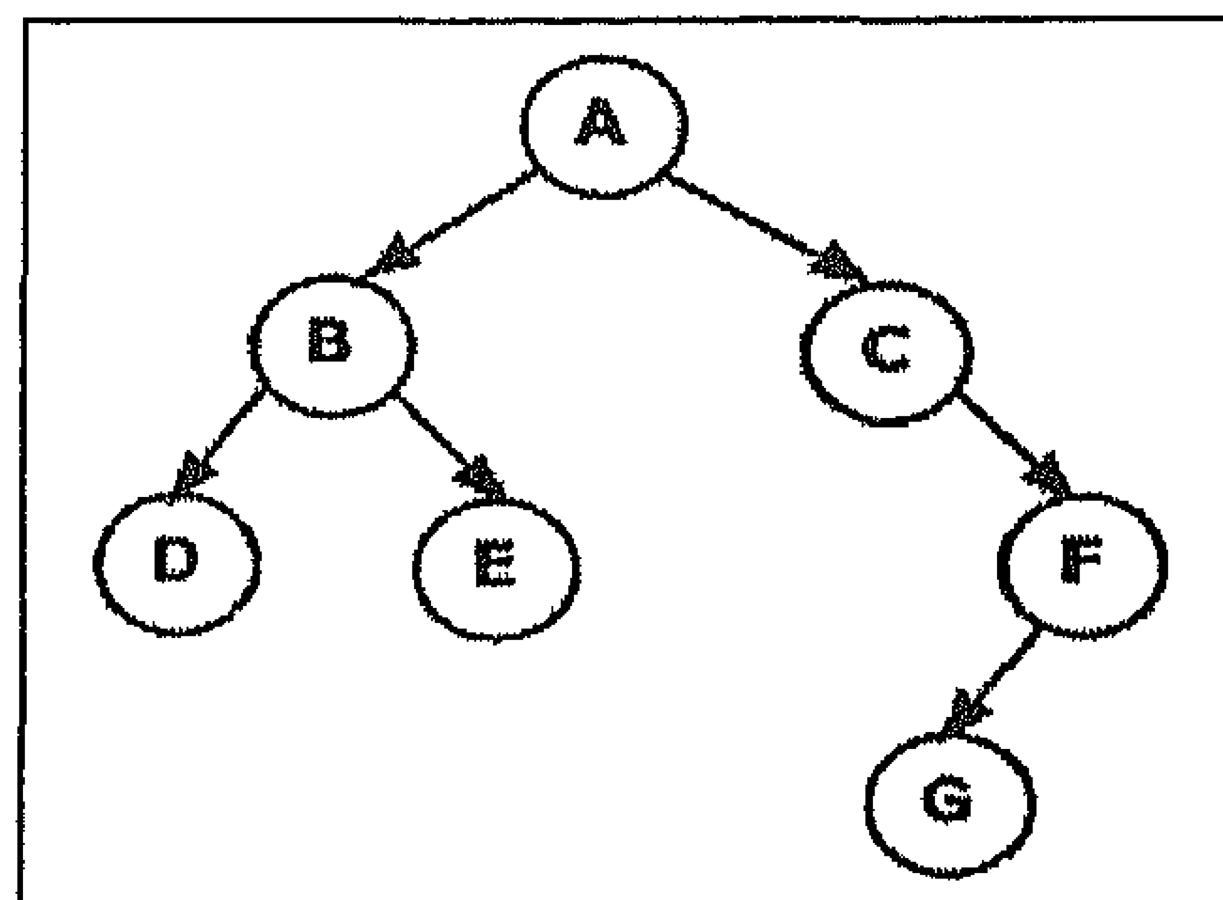
مثال (8)

الهيكل الشجري المتسع للمخطط المبين في الشكل (17) التالي:



الشكل (17)

هو كما يلي:



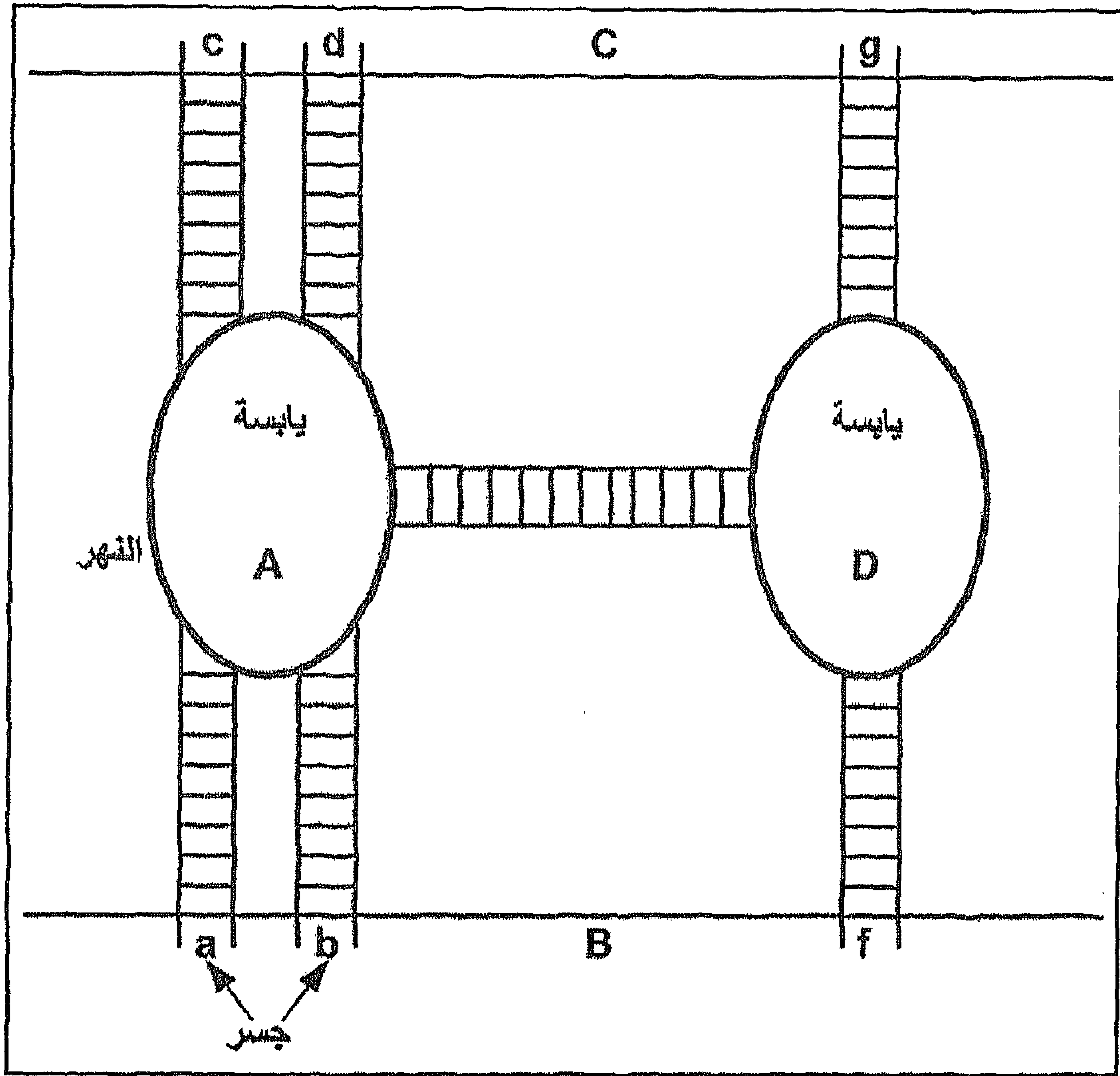


أسئلة التقويم الذاتي (3)

لخّص العلاقة بين الهياكل الشجرية والمخططات.

5. أمثلة وتطبيقات عملية

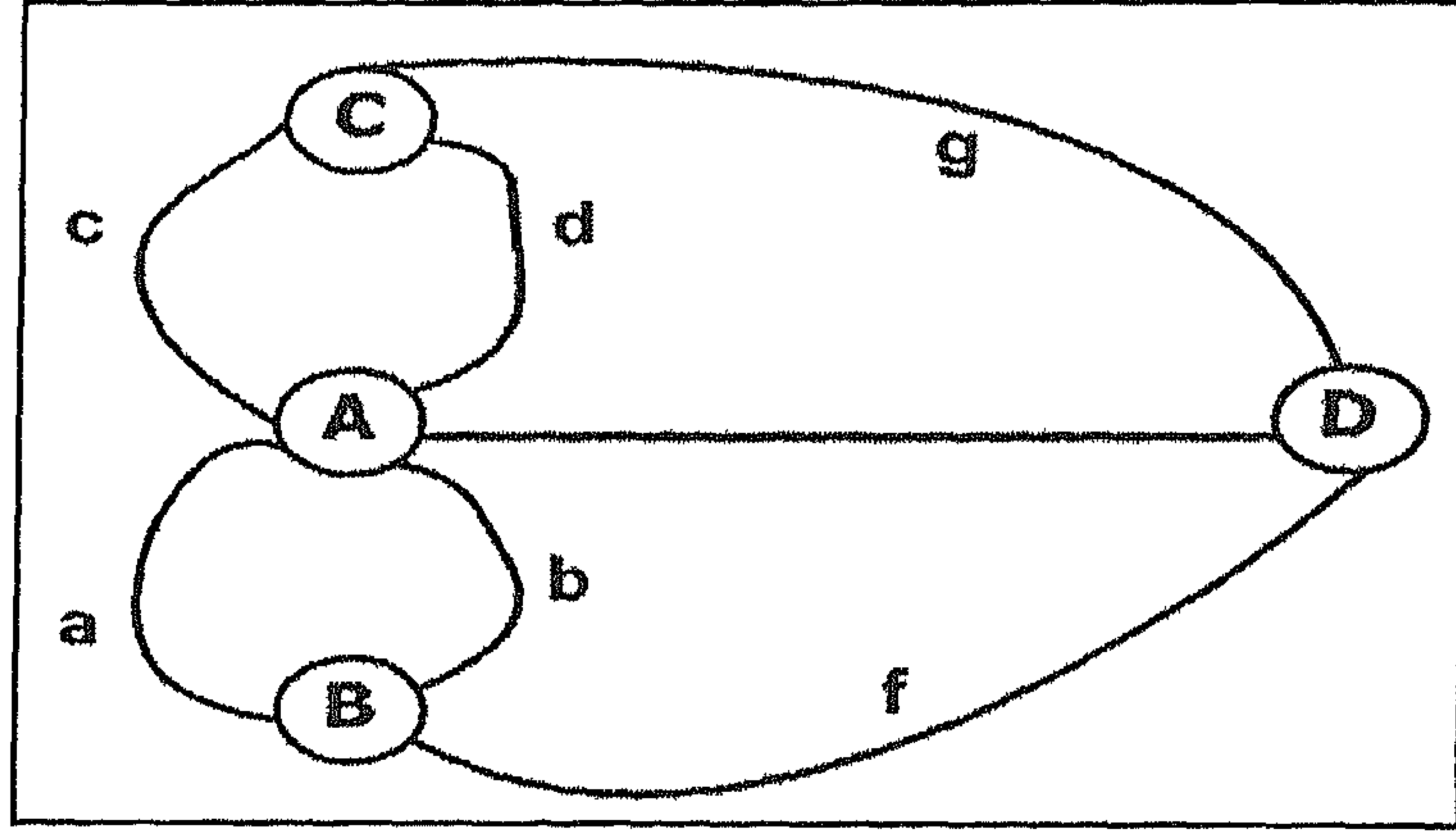
تطرقنـا، عزيـزي الدارس، خلال شـرحنا للمخططات عـن بعض التطبيقـات التي يمكن استخدامها فيها، وبالعودة إلى الوراء نجد أن فكرة استخدام المخططات قديمة منذ عام 1736، وذلك عندما اسـتخدمها العالم أويلر (Euler) لحل مسألة جسور كوينبرغ (Koenigsberg) حيث تقع هذه المدينة في بروسيا، ويمر في هذه المدينة نهر بريجال (Pregal) حول جزيرة نيفوف (Kneiphof)، وبعد ذلك ينقسـم النهر إلى نهرين وتفصل اليابسة بين هذين النهرين كما هو مبين في الشـكل (18) التالي:



الشكل (18)

وتتلخص مسـألة جسـور كوينبرغ بتحديد ما إذا كان من الممكن إذا بدأ شـخص المسير انطلاقاً من أي من مناطق اليابسة أن يمر عبر جميع الجسور ولمرة واحدة فقط وأن يعود إلى نقطة الانطلاق دون أن يمر على الجسر الواحد أكثر من مرة واحدة؟

وكان جواب العالم (بولسر) بالنفي، وذلك بعد أن حل هذه المسألة بتحويلها إلى مخطط حيث تمثل مناطق اليابسة فيه عناصره وتمثل الجسور حواف هذا المخطط كما هو مبين في الشكل (19):



الشكل (19)

وقد بين أويلر في حله لهذه المسألة، بأنه لكي يكون ممكناً المرور بجميع الجسور مرة واحدة والعودة إلى نقطة الإنطلاق فلا بد من أن تكون رتب جميع العناصر المكونة للمخطط زوجية.

ونعني برتبة العنصر، عدد الحواف التي تصل هذا العنصر. ومن المخطط في الشكل (19) نجد أن رتبة العنصر A تساوي 5 وأن رتبة كل من D, C, B تساوي 3 وهذا يعني أن جواب هذه المسألة بالنفي لكون رتب جميع عناصر المخطط فردية.

ومن التطبيقات العملية الأخرى للمخططات نورد ما يلي:

1. تحليل الدوائر الكهربائية. (Electric Circuit Analysis)
2. إيجاد أقصر الممرات. (Nding Shortest Path)
3. تخطيط المشاريع. (Project Planning)
4. تحديد المركبات الكيماوية. (Identification of Chemical Compounds)
5. التراكيب الرياضية. (All Mathematical Structures)
6. مجال اللغويات. (Linguistics)
7. مجال علم الاجتماع (Social Sciences)

حيث كنا قد تطرقنا، عزيزي الدارس، إلى استخدام المخططات في إيجاد أقصر الممرات بشيء من التفصيل.

6. الخلاصة

في هذه الوحدة تناولنا، عزيزي الدارس، موضوع المخططات والشبكات من حيث طرق تمثيلها، وإيجاد أقصر الممرات باستخدام المخططات، والعلاقة بين الهياكل الشجرية والمخططات، ودرسنا بعض الأمثلة والتطبيقات.

1. توجد طريقتان رئيستان لتمثيل المخططات داخل ذاكرة الحاسوب هما:
2. باستخدام القوائم المتصلة (Linked Representation).
3. باستخدام مصفوفة الجوار (Adjacency Matrix).

أما بالنسبة لإيجاد أقصر الممرات باستخدام المخططات فيعتبر هذا الجزء تطبيقاً عملياً للمخططات ويعالج موضوعات شتى ذات أهمية في حياتنا اليومية. وفي هذه الوحدة تجد برنامجاً مكتوباً بلغة باسكال لإيجاد أقصر الممرات بين نقطتين باستخدام المخططات.

وأذكرك هنا بالعلاقة بين الهياكل الشجرية والمخططات، جميع الهياكل الشجرية تعتبر مخططات بينما العكس غير صحيح، وذلك لوجود بعض القيود على الهياكل الشجرية والتي لا تنطبق على المخططات.

وفي نهاية هذه الوحدة تجد مثالاً على استخدام المخططات وكذلك بعض التطبيقات العملية عليها.

7. لمحة عن الوحدة الدراسية الحادية عشرة

تتناول الوحدة الحادية عشرة، وهي الأخيرة من هذا الكتاب، موضوعاً تشكل محتوياته جزءاً مهماً من العمليات الأساسية في علم الحاسوب، ألا وهو الفرز والبحث، حيث أن الفرز يعني عملية ترتيب البيانات حسب ترتيب معين إما تصاعدياً أو تنازلياً. ويمكن لهذه البيانات أن تكون عددية أو رمزية أو سلاسل رمزية.

أما البحث (الاستقصاء) فيعني عملية إيجاد موقع عنصر ما ضمن مجموعة من العناصر.

ومن الموضوعات التي تتناولها هذه الوحدة: مقدمة إلى الفرز والبحث، طرق الفرز، وطرق البحث.

يمكن لطريقة الفرز أن تكون داخلية (أي أنه يمكن استيعاب البيانات بالكامل داخل ذاكرة الحاسوب) أو خارجية (عندما لا يمكن استيعاب جميع البيانات داخل ذاكرة الحاسوب دفعة واحدة).

من طرق الفرز الداخلية التي سنتناولها في الوحدة التالية: الفرز التجزئي، والفرز السريع، والفرز المقيد.

أما طريقة الفرز الخارجية التي سوف نتطرق إليها تشمل الفرز بالدمج.
وبالنسبة لطرق البحث، سنتطرق إلى: البحث التتابعي، والبحث الثنائي، والبحث المفهرس.
وفي نهاية تلك الوحدة ستكتشف، عزيزي الدارس، قيمة المعلومات والمهارات العملية المتضمنة فيها ومجالات الحياة المختلفة التي يمكنك تطبيق هذه المعلومات والمهارات فيها.

8. إجابات التدريبات

تدريب (1)

القيمة الابتدائية للمصفوفات هي:

	1	2	3	4	5
DISTANCE	200	32767	32767	400	0
PATH	5	32767	32767	5	0
INCLUDED	false	false	false	false	true

حيث أن القيمة 32767 (تمثل هذه القيمة ما لا نهاية (∞) في الحاسوب) تعني عدم وجود حافة بين نقطة البداية وهذه النقطة. وعلى سبيل المثال نجد أسفل النقطة2 القيمة 32767 وهذا يعني عدم وجود حافة بين النقطة 5 والنقطة 2.

DISTANCE	200	1000	3185	400	0
PATH	5	1	1	5	0
INCLUDED	true	false	false	false	true

DISTANCE	200	1000	1821	400	0
PATH	5	1	4	5	0
INCLUDED	true	false	false	true	true

DISTANCE	200	1000	1410	400	0
PATH	5	1	2	5	0
INCLUDED	true	true	false	true	true

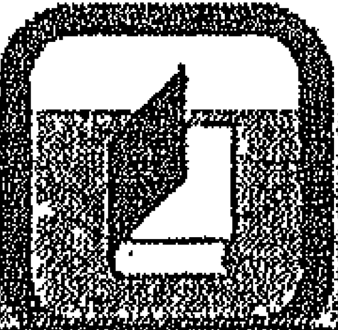
DISTANCE	200	1000	1410	400	0
PATH	5	1	2	5	0
INCLUDED	true	true	true	true	true

The path is: 5 → 1 → 2 → 3

ويمكنك عزيزي الدارس، الرجوع إلى المثال المحلول في متن المقرر (مثال 6) لإيجاد أقصر الممرات لمتابعة كيفية الحصول على قيم هذه المصفوفات.

9. مسرد المصطلحات

- أقصر ممر Shortest Path
- الحواف Edges
- الشبكات Networks
- **المخططات Graphs**: نوع من تراكيب البيانات بحيث يتكون المخطط من مجموعة من النقاط أو الرؤوس (Vertices) ومجموعة من الحواف (Edges). ويمكن تمثيل المخطط G على النحو التالي $G = (V, E)$ بحيث تمثل V مجموعة من رؤوس المخطط أو نقاطه وتمثل E مجموعة الحواف (Edges) .
- **الهيكل الشجري المتسع Spanning Tree**: هيكل شجري ينتج من المخطط المتصل بعد إزالة الحواف التي تعمل دورات في المخطط.



10. المراجع

1- Kruse, Robert L., Data Structures and Program Design. Englewood Cliffs (USA): Prentice-Hall, 1984.

2- Lipschutz, Seymour, Theory and Problems of Data Structures. New York: McGraw-Hill, 1986.

3- Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison Wesley, 1999.

4- Lewis, T. G.; Smith, M.Z., Applying Data Structures. Atlanta (USA): Houghton Mifflin, 1976.

5- Tenenbaum, Aarom M.; Augenstein, Moshe J., Data Structures Using Pascal. Englewood Cliffs (USA): Prentice-HALL, 1981.



الفرز والبحث (Search & Sort)

محتويات الوحدة

1. المقدمة

1.1 تمهيد

أهلاً بك، عزيزي الدارس، إلى رحاب الوحدة الحادية عشرة وهي الأخيرة من كتاب "تركيب البيانات وتصميم الخوارزميات"، والتي تعالج موضوعين مهمين في الحياة العملية هما الفرز والبحث. تبدأ هذه الوحدة بموضوع الفرز والذي يعني ترتيب مجموعة من العناصر حسب قيمة مفتاح معين. وقد يكون الترتيب إما تصاعدياً أو تنازلياً لقيمة المفتاح. ونظراً لكثرة استخدام عمليات الفرز في الحياة العملية، سنعالج في ثنايا الوحدة طرق الفرز المختلفة الداخلية والخارجية. ومع أن النتيجة واحدة لجميع طرق الفرز المختلفة إلا أن الاختلاف بينها ينشأ من طريقة معالجة البيانات في المراحل البينية. كذلك فإن هذه الطرق يختلف بعضها عن البعض الآخر من حيث الكفاءة. فبعضها أكثر ملاءمة من الآخر لعدد قليل من البيانات بينما البعض الآخر أكثر ملاءمة عندما يكون حجم البيانات المراد فرزها كبيراً.

لقد تناولنا في الوحدة الثانية (مبادئ تحليل الخوارزميات) ثلاثة أنواع من طرق الفرز الداخلية هي الفرز الفقاعي والفرز الانتقائي (الاختياري) والفرز الإدخالي. وسنتناول في هذه الوحدة ثلاث خوارزميات فرز أخرى هي الفرز التجزئي والفرز السريع والفرز المقيد. ومن طرق الفرز الخارجية سنتناول بشيء من التفصيل طريقة الفرز بالدمج.

ننتقل بعد ذلك إلى موضوع آخر هو طرق البحث. ونعالج في هذا الموضوع ثلاث طرق رئيسة هي البحث التتابعي والبحث الثنائي والبحث المفهرس بأنواعه الثلاثة (المفهرس التتابعي وباستخدام الهياكل الشجرية الثنائية وباستخدام الهياكل الشجرية نوع – B)، وبالنسبة لموضوع الفهرسة باستخدام الهياكل الشجرية نوع- B، كنا قد عالجنا هذا النوع من الهياكل الشجرية في الوحدة السابقة بشيء من التفصيل. وقد حاولنا في هذه الوحدة الإكثار من الأمثلة والتدريبات ذات العلاقة بموضوعات الوحدة وحلولها الأنموذجية.

ولتدعيم المادة والمفاهيم المختلفة الواردة في هذه الوحدة يمكنك تطبيق الأمثلة والتدريبات على الحاسوب بكتابة برامج لطرق الفرز والبحث المختلفة وتنفيذها على أجهزة الحاسوب المتوفرة لديك.

أهلاً بك مرة أخرى إلى رحاب هذه الوحدة وأرجو أن تستمتع بدراستها وأن تنتفع بموضوعاتها. ونحن بانتظار مشاركتك واستفساراتك حولها.

2.1 أهداف الوحدة

ينتظر منك، عزيزي الدارس، بعد قراءة هذه الوحدة أن تكون قادراً على أن:

1. توضح أهمية الفرز والبحث عند معالجة البيانات.
2. تميز طرق الفرز المختلفة عن بعضها.
3. تميز طرق البحث المختلفة عن بعضها.
4. تقوم ببناء الخوارزميات والبرامج اللازمة الخاصة بفرز البيانات والبحث عنها.
5. تحلل خوارزميات الترتيب المختلفة، وتقارن فيما بينها.

3.1 أقسام الوحدة

إن الأقسام الرئيسة الثلاثة المكونة لهذه الوحدة تتبع قائمة الأهداف التعليمية المشار إليها أعلاه. حيث يرتبط القسم الأول وهو مقدمة إلى الفرز والبحث بشكل واضح بالهدف التعليمي الأول في قائمة الأهداف المشار لها سابقاً.

أما القسم الثاني من هذه الوحدة وهو وطرق الفرز، فيرتبط ارتباطاً مباشراً بالهدف التعليمي الثاني، والرابع والخامس ويطرح موضوع طرق الفرز الداخلية والخارجية.

وأما القسم الثالث من هذه الوحدة فتتعرف منه على طرق البحث المختلفة وتستطيع بعد هذا القسم التمييز بين هذه الطرق. ويرتبط هذا القسم بالهدف الثالث وجزء من الهدف الرابع.

4.1 القراءات المساعدة

عزيزي الدارس حاول الانتفاع ما أمكن بالقراءات التالية نظراً لاتصالها المباشر بموضوع هذه الوحدة. ولا شك أن في ذلك تعميقاً لفهمك واستيعابك للموضوع:

1. Kruse, Robert L., Data Structures and Program Design. Englewood Cliffs (USA): Prentice-Hall, 1984.

2. Lipschutz, Seymour, Theory and Problems of Data Structures. New York: McGraw-Hill, 1986.

2. مقدمة إلى الفرز والبحث

لا شك، عزيزي الدارس، أنك قد تواجه في حياتك اليومية الكثير من الأمور التي تتطلب حفظ المعلومات والبحث عنها. وهذا قد يقتضي منك حفظ هذه المعلومات أو البيانات وفق ترتيب معين يسهل عليك عملية البحث عن قيمة معينة ضمنها. وفي هذه الأيام تعتبر عمليات الفرز والبحث من العمليات الأساسية في مجال علم الحاسوب، حيث تم تطوير خوارزميات عدة لإنجاز عمليات البحث والفرز بكفاءة عالية. ففي السنوات الماضية، كان أكثر من نصف وقت استخدام الكثير من الحواسيب التجارية يضيع في عمليات الفرز والبحث. أما في الوقت الحالي، ونتيجة لتطوير خوارزميات متطورة ذات كفاءة عالية لتنظيم البيانات داخل ذاكرة الحاسوب، فقد أصبح من غير الضروري حفظ البيانات مرتبة مما يوفر الكثير من الوقت، حيث يمكن إجراء عمليات الفرز والبحث عند الحاجة إلى الحصول على البيانات مرتبة حسب حقل من الحقول.

ويقصد بالفرز عملية ترتيب البيانات قيد البحث، حسب ترتيب معين، إما تصاعدياً أو تنازلياً. ويمكن للترتيب أن يكون على قيم عددية أو رمزية، كأن نرتب قائمة من الطلبة حسب أرقامهم، أو معدلاتهم، أو أسمائهم.

أما البحث (الاستقصاء) فيشير إلى عملية إيجاد موقع عنصر معين ضمن مجموعة من القيم. فإن لم يكن ذلك العنصر موجوداً يمكن طباعة إشعار بذلك. ويمكن البحث عن قيمة عددية أو قيمة رمزية كذلك.

وتبرز أهمية الفرز والبحث عند معالجة مجموعة كبيرة من السجلات. فعلى سبيل المثال، يتم في بداية كل فصل دراسي طباعة أرقام جميع الطلبة المسجلين في الجامعة وأسمائهم وذلك لتوزيع هذه القوائم على المرشدين. كما يتم أيضاً طباعة قوائم بأسماء الطلبة المسجلين للمقررات الدراسية المختلفة في الجامعة لتوزيع هذه القوائم على مدرسي هذه المقررات لمعرفة أسماء الطلبة المسجلين في المقررات التي يدرسونها. ومثل هذه القوائم تأتي عادة مرتبة حسب الرقم الجامعي للطالب. وبذلك يمكن للمرشد أن يعرف الطلبة الذين يقوم بإرشادهم ويتابع سيرتهم الأكاديمية. وكذلك فإن مدرس المقرر يمكنه البحث عن اسم طالب معين في القوائم المرتبة حسب الرقم الجامعي بشكل سهل وأفضل. وبالمثل نستطيع أن نستخدم خوارزميات بحث فعالة وسريعة إذا كانت القوائم مرتبة، بينما لا نستطيع ذلك فيما إذا كانت غير مرتبة. من هنا يتبين لنا أهمية الفرز والبحث حيث

أننا من الناحية العملية نحتاج إلى عمليات الفرز والبحث في جميع الأعمال التي تتطلب الاحتفاظ بالبيانات وسرعة الوصول إليها. ومن الأمثلة على ذلك الأنظمة الكثيرة المطبقة على الحاسوب مثل نظام اللوازم، ونظام القبول والتسجيل، ونظام المالية، ونظام المكتبة، وما إلى ذلك.

وتتطلب عمليات الفرز والبحث أن ننشئ خوارزمية ملائمة تؤثر على اختيارها عدة عوامل منها ما يلي:

أولاً: الوقت الذي يحتاجه المبرمج لكتابة برنامج ينفذ الخوارزمية بلغة برمجة معينة.

ثانياً: المدة التي يستغرقها تنفيذ البرنامج (Execution time).

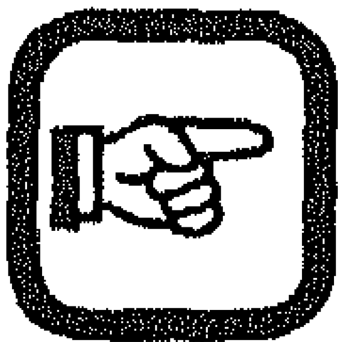
ثالثاً: مساحة الذاكرة التي يحتاجها تنفيذ الخوارزمية.

رابعاً: خواص البيانات مثل عددها ونوعها.

فإذا كان عدد العناصر المراد فرزها في الملف قليلاً (100-50 على سبيل المثال)، فلا داعي للبحث عن خوارزمية ذات كفاءة عالية قد تحتاج كتابتها إلى جهد أكبر من قبل المبرمج. ففي مثل هذه الحالة تفي أي خوارزمية بالغرض المطلوب.

وأما إذا كان عدد العناصر المراد فرزها كبيراً جداً وخصوصاً عندما يكون طول كل سجل من هذه العناصر كبيراً، فإنه من المنطقي أن نحاول إيجاد خوارزمية للفرز وخوارزمية للبحث يستغرق كل منهما أقل وقت ممكن عند التنفيذ على جهاز الحاسوب.

وفي بعض الأحيان نحتاج إلى استخدام عمليات الفرز والبحث على ملف من السجلات، حيث يتكون كل سجل منها من عدد من الحقول، ويستعمل أحد هذه الحقول للتمييز بين السجلات المختلفة. ويطلق على مثل هذا الحقل بالمفتاح (Key) . وبما أنه يمكن أن يستخدم الملف نفسه لأغراض مختلفة (كما هو الحال في قواعد البيانات)، فمن الممكن استعمال حقول مختلفة للتمييز بين السجلات وذلك حسب التطبيق الذي استخدم الملف من أجله.



مثال (1)

يمكن عمل ملف خاص بالطلبة المسجلين في التعليم المفتوح، بحيث يحتوي سجل الطالب على بعض المعلومات مثل رقم الطالب الجامعي، واسم الطالب، وعنوانه، بالإضافة إلى معلومات أخرى. في مثل هذه الحالة يمكن اعتبار رقم الطالب الجامعي على أنه المفتاح

الذي يميز السجلات بعضها عن البعض الآخر، بحيث أن رقم كل طالب يختلف عن رقم أي طالب آخر.

ومن الممكن في بعض الأحوال أن تجد الرغبة في إيجاد سجل طالب معين لا تعرف رقمه، بل تعرف اسمه. وفي مثل هذه الحالة يستعمل اسم الطالب كمفتاح بحيث يتم استقصاء ملف الطالب على الاسم المذكور لإيجاد معلومات عن الطالب. وفي تطبيق آخر قد تحتاج إلى إيجاد رقم الطالب على عنوان معين، في هذه الحالة يستخدم العنوان على أنه المفتاح الذي يستخدم في عملية الاستقصاء. من هنا نجد أهمية تحديد المفتاح (Key) في عمليات الفرز والبحث التي تستخدم بكثرة في معظم عمليات معالجة البيانات.

يمكن القول، إذن، بأن هناك فائدة كبيرة ومهمة للفرز هي المساعدة في إنجاز عمليات البحث بفعالية عالية، كما سنرى عند مناقشة طرق البحث، إذ أنه عند توفر البيانات بشكل مرتب يمكن تنفيذ عمليات البحث الثنائي التي تستغرق وقتاً قليلاً مقارنة مع طرق البحث الأخرى. بالإضافة إلى أن البيانات المفروزة أفضل للعرض على المستخدم. تصور، عزيزي الدارس، أن برنامجاً لحساب الرواتب يعطينا قائمة بأسماء الموظفين ورواتبهم بدون ترتيب، لا شك وأننا سنفضل برنامجاً آخر يعطينا هذه القائمة مرتبة حسب الترتيب الأبجدي للموظفين أو وفق مقدار رواتبهم.

3. طرق الفرز

يمكن تقسيم الفرز إلى نوعين رئيسين هما الفرز الداخلي والفرز الخارجي. يستخدم الفرز الداخلي إذا أمكن استيعاب جميع سجلات الملف في ذاكرة الحاسوب الداخلية (الرئيسة) السريعة. وبذلك يمكن تنفيذ البرنامج على جميع السجلات، وهذا النوع من الفرز يستعمل عندما يكون عدد السجلات معقولاً ويمكن استيعابه بواسطة ذاكرة الحاسوب الرئيسة في الوقت نفسه.

وفي حالة وجود عدد كبير جداً من السجلات في الملف المراد فرزه، فإنه من غير الممكن استيعاب جميع سجلات الملف في ذاكرة الحاسوب الرئيسة في الوقت نفسه. لذلك يتم الاستعانة بوسائط التخزين المساعدة، مثل الأقراص والأشرطة المغناطيسية وما إلى ذلك. وفي هذا القسم سنتطرق، إلى طرق الفرز الداخلية ثم إلى الفرز الخارجي.

1.3 طرق الفرز الداخلية (Internal Sorting Methods)

نتناول في هذا البند ثلاثة أنواع من طرق الفرز الداخلية والتي تتطلب وجود جميع السجلات أو البيانات المراد فرزها في الذاكرة السريعة للحاسوب. تذكر، عزيزي الدارس، أن طرق الفرز الأخرى التي ناقشناها في الوحدة الثانية كانت جميعها من طرق الفرز الداخلية.

1.1.3 الفرز التجزئي (Shell Sort)

سمي هذا النوع من أنواع الفرز بهذا الاسم نسبة إلى مصممه (D.L.Shell)، ويعتبر فرزاً إدخالياً معدلاً.

وبشكل عام فإن المرحلة الأولى من الفرز التجزئي تتضمن تقسيم القائمة إلى مجموعة من القوائم الجزئية حيث يتم ترتيبها باستخدام الفرز الإدخالي.

وفي المرحلة الثانية يتم تقسيم المصفوفة إلى عدد أقل من القوائم الجزئية الجديدة بحيث يتم ترتيبها باستخدام الفرز الإدخالي (انظر الوحدة الثانية) مرة أخرى. ويمكن الاستمرار في عملية التقسيم إلى قوائم جزئية وترتيبها باستخدام الفرز الإدخالي وذلك حسب رغبة المبرمج وإلى المدى الذي يشاء.

وفي المرحلة الأخيرة من عملية الفرز يتم اعتبار المصفوفة بالكامل على أنها مصفوفة واحدة وبذلك يتم ترتيب القائمة حسب الأصول.

فلو فرضت أنه تم تقسيم المصفوفة في المرحلة الأولى إلى K من القطاعات، حيث يفضل أن تكون قيمة K من القيم الأولية (Prime numbers)، وأن عدد العناصر في المصفوفة يساوي N، فإن القوائم الجزئية التي تحصل عليها هي كالتالي:

القائمة الأولى: A[0], A[K], A[2*K], ...,

القائمة الثانية: A[1], A[K+1], A[2*K+1], ...,

......

القائمة K: A[K-1], A[2*K-1], A[3*K-1], ...,

لاحظ، عزيزي الدارس، أن K تمثل عدد القوائم وبنفس الوقت مقدار بعد كل عنصر في قائمة ما عن العنصر الذي يليه في نفس القائمة.

وبما أنه يتم فرز كل قائمة جزئية على حدة فإنه بعد الجولة الأولى للخوارزمية تصبح القائمة مرتبة جزئياً.

وفي الجولة الثانية يتم اختيار قيمة جديدة لــ K أقل من القيمة التي تم اختيارها في المرة الأولى. وبذلك يزداد عدد العناصر في القوائم الجديدة، ويقل عدد هذه القوائم عن التي تم الحصول عليها في الجولة الأولى.

وتستمر هذه العملية حتى تصبح قيمة K تساوي 1، ثم نطبق خوارزمية الفرز الإدخالي مرة أخرى فنحصل على المصفوفة الأصلية مرتبة (مفروزة).

ولعلك، عزيزي الدارس، تتساءل ما الحكمة من هذه الخوارزمية إذا كنا سننفذ خوارزمية الفرز الإدخالي على كل المصفوفة في نهاية الأمر. أي لم كل هذه الخطوات السابقة؟

في الحقيقة هنالك فائدتان لهذا العمل:

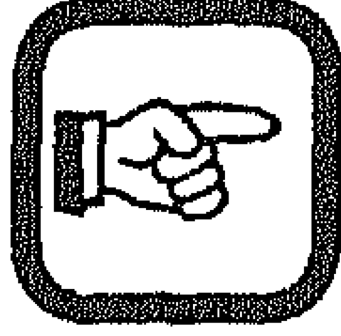
الفائدة الأولى:

إن ترتيب قائمة تبتعد عناصرها عن بعضها البعض K من العناصر يزيد من احتمالية أن يصل العنصر بسرعة إلى موقعه النهائي في المصفوفة. لفهم ذلك تصور، عزيزي الدارس، وجود عنصر كبير جداً في بداية المصفوفة (A[0]) هذا يعني أنه في المرحلة الأولى سيقارن مع A[K] ثم مع A[2K]، مما يعني أنه سينتقل بسرعة إلى نهاية المصفوفة.

الفائدة الثانية:

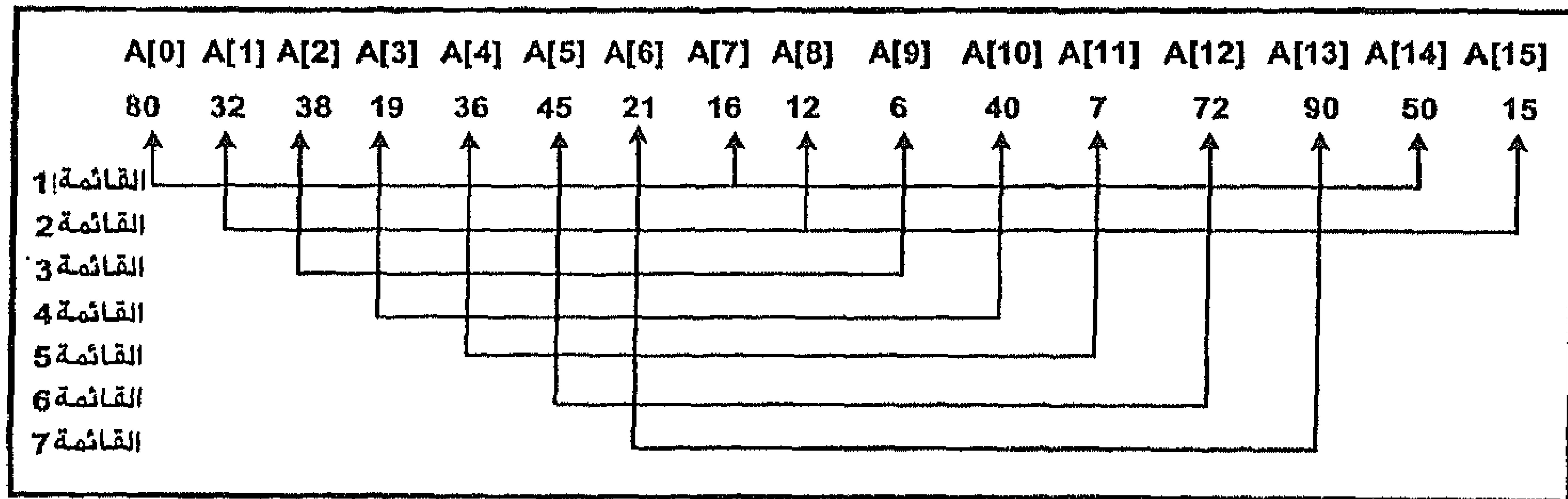
تذكر، عزيزي الدارس، أن خوارزمية الفرز الإدخالي فعالة جداً إذا كانت المصفوفة شبه مرتبة مسبقاً (ينصح الدارس بمراجعة الخوارزمية في الوحدة الثانية من المقرر)، وذلك لأن

العنصر الموجود في مكانه الصحيح لن يقارن مع بقية العناصر. وعليه ولكون كل مرحلة من مراحل خوارزمية الفرز التجزئي ينتج عنها مصفوفة مرتبة جزئياً، فهذا يعني أن تطبيق خوارزمية الفرز الإدخالي لن يكون مكلفاً (من حيث عدد عمليات المقارنة والاستبدال) في الواقع في كل مرة تطبق خوارزمية الفرز الإدخالي فهي تطبق على مصفوفة أكثر ترتيباً من ذي قبل، وبذلك تكون أقل تكلفة.



مثال (2)

افرض أنك ترغب بترتيب القائمة التالية باستخدام طريقة الفرز التجزئي (الشكل (1)):



الشكل (1)

كما هو مبين في الشكل أعلاه، تم تقسيم عناصر المصفوفة إلى (K = 7) قوائم جزئية (حيث أن 7 يمثل عدد أولى) وذلك كما يلي:

القائمة الأولى:	80 ،16 ،50
القائمة الثانية:	32 ،12 ،15
القائمة الثالثة:	38 ،6
القائمة الرابعة:	19 ،40
القائمة الخامسة:	36 ،7
القائمة السادسة:	45 ،72
القائمة السابعة:	21 ،90

أما عن كيفية الحصول على هذه القوائم فقد أشرت إلى ذلك سابقاً، ويمكنك، عزيزي الدارس، حساب مكونات عناصر القائمة الأولى كالتالي:

$A[0] = 80$

$A[K] = A[7] = 16$

$A[2K] = A[14] = 50$

وهنا تتوقف عن إضافة أي عنصر جديد إلى هذه القائمة وذلك لكون العنصر [3K=21] هو أكبر من حجم المصفوفة الكلية، وهو 16 عنصر.

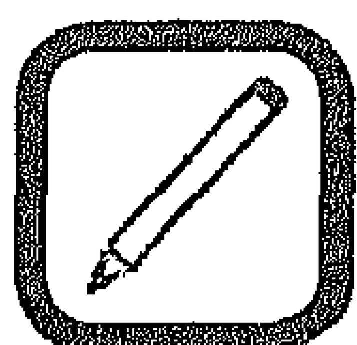
وبالطريقة نفسها يمكنك الحصول على المجموعة الثانية والتي تتكون من العناصر الثلاث التالية:

A[1] = 32

A[1 + K] = A[1 + 7] = A[8] = 12

A[1 + 2K] = A[1 + 14] = A[15] = 15

وبما أن [1 + 2K] يساوي 15 (أي عدد العناصر في المجموعة الكلية) فلا توجد عناصر أخرى في هذه المجموعة.



تدريب (1)

كرر الخطوات السابقة وذلك لبيان كيف تحصل على عناصر القوائم 3 إلى 7.

لإكمال المرحلة الأولى في المثال (2) السابق ترتب القوائم الجزئية منفصلة وذلك باستخدام الفرز الإدخالي، وبترتيب القائمة الجزئية الأولى تصبح عناصرها على النحو التالي:

List[0]=16 List[7]=50 List[14]=80

وبتكرار عملية الفرز الإدخالي على جميع القوائم الجزئية كل على حدة تصبح القوائم الجزئية مرتبة، وتبدو القائمة الكلية على النحو التالي:

	A[0]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	16	12	6	19	7	45	21	50	15	38	40	36	72	90	80	32

ولبدء مرحلة جديدة نختار قيمة أخرى لـ (K) مختلفة أصغر من القيمة السابقة، (ولذلك فإن عدد القوائم سيكون أقل، ولكن عدد العناصر في كل قائمة سيكون أكبر)، وتكرر الخطوات السابقة. وعلى سبيل المثال إذا اخترت K = 3 (حيث أن 3 عدد أولي أيضاً)، فإن محتويات القوائم الجزئية تصبح على النحو التالي:

محتويات القائمة الأولى هي:

```
A[ 0 ]            = 16
A[K ]  = A[ 3 ]  = 19
A[2K ] = A[ 6 ]  = 21
A[3K ] = A[ 9]   = 38
A[4K ] = A[ 12]  = 72
A[5K ] = A[ 15]  = 32
```

القائمة الثانية: 12, 7, 50, 40, 90

القائمة الثالثة: 6, 45, 15, 36, 80

بعد ذلك ترتب هذه القوائم الجزئية لنحصل على النتيجة التالية:

القائمة الأولى: 16, 19, 21, 32, 38, 72

القائمة الثانية: 7, 12, 40, 50, 90

القائمة الثالثة: 6, 15, 36, 45, 80

ومن ثم تبدو القائمة الكلية كالتالي:

16, 6, 7, 19, 12, 15, 21, 40, 36, 32, 50, 45, 38, 90, 80, 72

لاحظ، عزيزي الدارس، أن هذه القائمة مرتبة جزئياً، وبالتالي فإن تطبيق خوارزمية الفرز الإدخالي عليها (أي أخذ K = 1 هذه المرة) لن يكون مكلفاً، لأن كثيراً من العناصر في مكانها الصحيح، أو على الأقل بالقرب منه.

ونكرر عملية التجزئة حتى نصل إلى قيمة K = 1 وبتطبيق خوارزمية الفرز وبعدها نحصل على القائمة الرئيسة مرتبة ترتيباً تصاعدياً كما هو مبين:

16, 6, 7, 12, 15, 16, 21, 32, 36, 38, 40, 45, 50, 72, 80, 90

ويجب أن تكون القيمة النهائية لــ K في (Shell Sort) تساوي 1، وهذا يعني أن القائمة بمجملها سوف تترتب بوصفها قائمة جزئية واحدة.

ويمكن كتابة دالة الفرز التجزئي بلغة سي++ على النحو التالي:

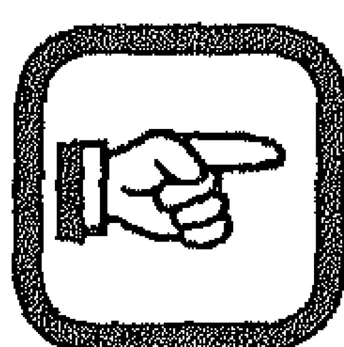
```
void ShellSort (ElementArray& A,int numVals)
// A هي المصفوفة التي نريد ترتيبها
// numVals هو عدد العناصر في المصفوفة
```

```
{int K;
k=numVals/2;
    K هو عدد القوائم وبنفس الوقت مقدار بعد كل عنصر عن الذي يليه//
while (K > 0)
    { // K استدع الخوارزمية SegmentedInsertion لترتيب القائمة
  SegmentedInsertion(A, numVals, K);
        K /=2;
// K وأعد الكرة قلل قيمة
        }
}
```

إن كل ما تعمله الخوارزمية ShellSort هو تحديد قيمة K، ثم تستدعي خوارزمية SegmentedInsertion. وما هي إلا صورة معدلة لخوارزمية الإدخال التي درسناها في الوحدة الثانية من المقرر. وقد تم تعديلها بحيث تقارن كل عنصر بالعنصر الذي يسبقه بـ K من العناصر وذلك حتى يتم ترتيب القوائم على مراحل. لاحظ، عزيزي الدارس، أن آخر قيمة لـ K ستكون واحداً. وإليك الآن خوارزمية Segmented Insertion.

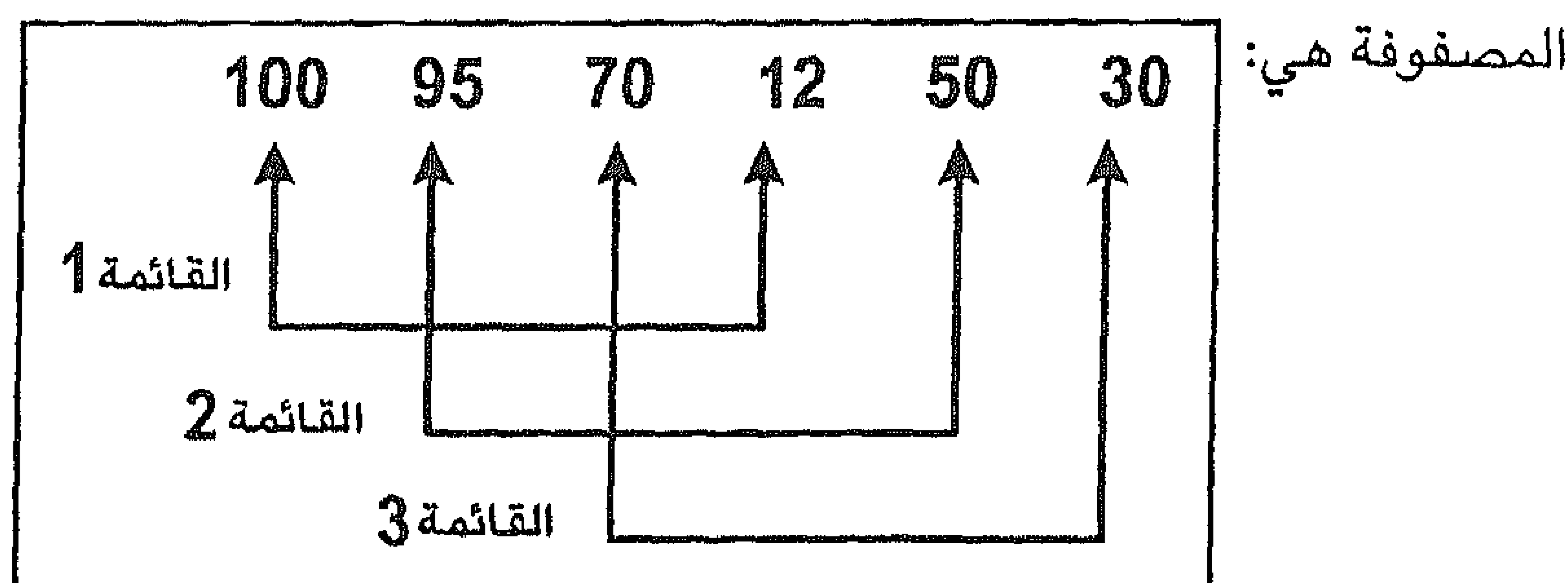
```
void SegmentedInsertion (ElementArray& A,int n, int K)
 {int temp,j,l;
 for(l=K+1;l<=n;l++)
  {j=l-K;
   // زيشير إلى العنصر السابق في القائمة
   while (j>0)
    {if (A[j+K]<A[j])
   // إذا العنصران المتتاليان في القائمة مرتبين بشكل غير صحيح، بدل أماكنهما
     {temp=A[j+K];
      A[j+K]=A[j];
      A[j]= temp;
      j=j-K;
     }
     else
     j=0;
   // العنصر A [j+K] أكبر من العنصر A[j]لذلك يجب وقف عملية المقارنة
    }
  }
 }
```

المثال (3) التالي يوضح كيف تعمل خوارزمية Segmented Insertion



مثال (3)

لنأخذ N = 6 (عدد القيم)، وبذلك تكون قيمة K أول مرة 3، وعليه فإن المصفوفة تقسم إلى ثلاث قوائم، المسافة في القائمة بين كل عنصر والذي يليه هي 3 . لنفرض أن عناصر المصفوفة هي:



القائمة الأولى:	12	100
القائمة الثانية:	50	95
القائمة الثالثة:	30	70

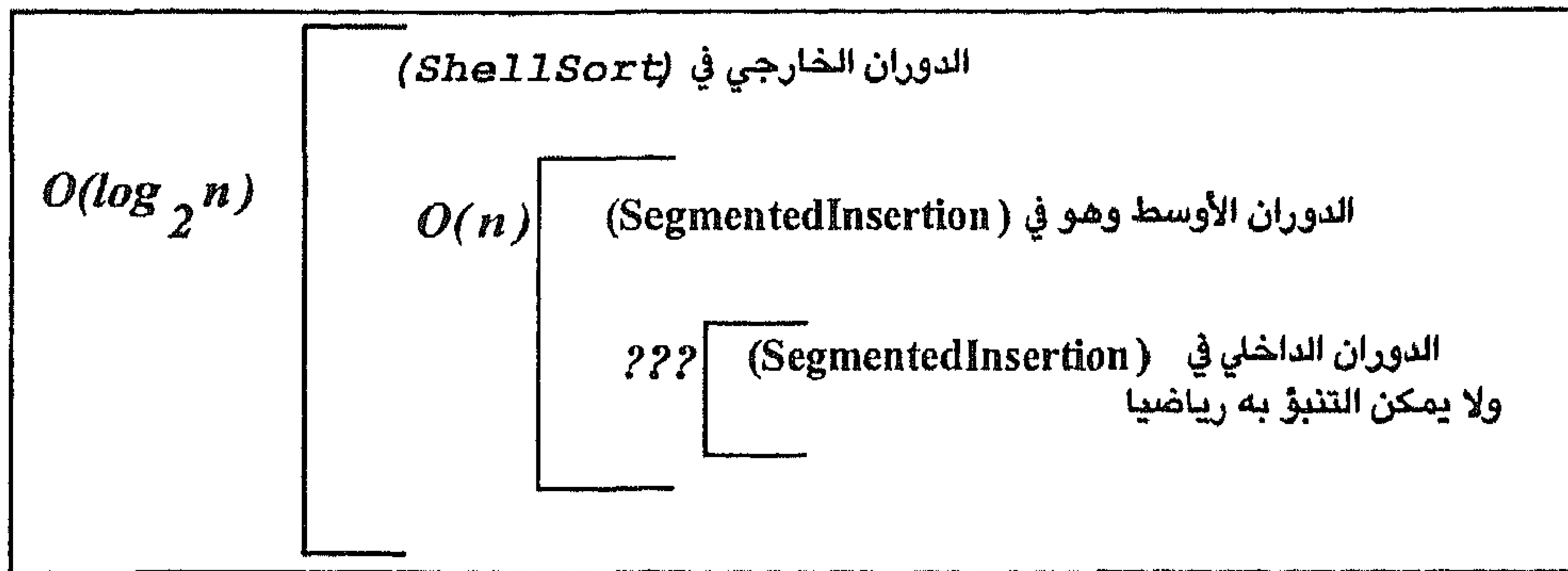
وعليه فإن استدعاء Segmented Insertion سينتج عنه ترتيب القوائم أعلاه كما يلي:

القائمة الأولى:	100	12
القائمة الثانية:	95	50
القائمة الثالثة:	70	30

تحليل خوارزمية الفرز التجزئي (Shell Sort)

لنحلل الآن، عزيزي الدارس، خوارزمية الفرز التجزئي باستخدام ترميز O الكبيرة. لاحظ أن الدوران الخارجي (الموجود في خوارزمية (Shell Sort) يتكرر $\log_2 n$ من المرات (وذلك لأننا نقسم K على 2 في كل مرة).

وأن الدوران الخارجي (for loop) في خوارزمية (Segmented Insertion) هو O(n). لكن ماذا عن الدوران الداخلي (while loop) في هذه الخوارزمية؟ كم مرة سيتكرر؟ يعتمد عدد مرات تكرار هذا الدوران على طبيعة بيانات المصفوفة، ولا يمكن التنبؤ به بشكل رياضي. والشكل (2) التالي يوضح ما ناقشناه لغاية الآن:



الشكل (2)

وبالمثل نستطيع أن نستخدم خوارزميات بحث فعالة وسريعة إذا كانت القوائم مرتبة بينما لا نستطيع ذلك إذا لم تكن هذه القوائم مرتبة.

إن خوارزمية الفرز التجزئي مثال جيد على الخوارزميات التي تتحدى التحليل الرياضي الكامل، ولمثل هذا النوع من الخوارزميات لا بد من دراسة الخوارزمية دراسة عملية باستخدام الحاسوب، وقياس عدد مرات تكرار الدوران الداخلي على بيانات عشوائية. وقد أجريت دراسات عديدة على هذه الخوارزمية. (انظر الجدول (1)) التالي:

جدول (1) نتائج الدراسات التي أجريت على خوارزمية Shell Sort

حجم عينة البيانات المولدة عشوائيا	معدل عدد مرات المقارنة	عدد المقارنات معبر عنه بـ $N \times (\log_2 N)^2$	عدد المقارنات معبر عنه بالحد $N^{(3/2)}$
1,000	14,702	$0.148 \times N \times (\log_2 N)^2$	$0.465 \times N^{(3/2)}$
3,000	57,958	$0.145 \times N \times (\log_2 N)^2$	$0.353 \times N^{(3/2)}$
5,000	109,065	$0.144 \times N \times (\log_2 N)^2$	$0.309 \times N^{(3/2)}$

يتضح من الجدول أن الخوارزمية هي O (log2 n)2 أو $O(N^{(3/2)})$.

2.1.3 الفرز السريع (Quick Sort)

يعتبر هذا النوع من أسرع أنواع الفرز الداخلي من حيث السرعة عند تطبيقه على القوائم الكبيرة. ويتلخص الفرز السريع باختيار عنصر معين من العناصر المراد ترتيبها، ومن ثم تقسيم العناصر المتبقية إلى مجموعتين، تعتبر كل مجموعة منهما مصفوفة جزئية جديدة.

وتمتاز المجموعة الأولى من هاتين المجموعتين بأن قيم جميع عناصرها أقل من العنصر الذي تم اختياره كقطب (Pivot). أما المجموعة الثانية فتشمل جميع العناصر الأكبر من

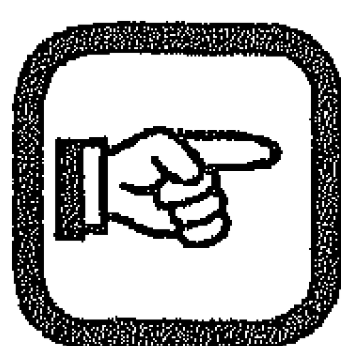
القيمة المختارة (Pivot). ويوضع القطب (Pivot) بين هاتين المجموعتين. وهكذا نكون قد رتبنا القائمة ترتيباً جزئياً.

وبتطبيق الفرز السريع على عناصر المجموعتين الأولى والثانية مرة أخرى كل على حدة تترتب القائمة أكثر فأكثر، وتعاد الكرة إلى أن تترتب القائمة بشكل كامل.

ويلاحظ أن في كل خطوة من خطوات الفرز السريع تستبدل مسألة فرز قائمة طويلة من العناصر بفرز قائمتين جديدتين كل منهما أصغر من القائمة الأصلية.

وباختصار تعمل خوارزمية الفرز السريع QuickSort كما يلي:

1. يتم اختيار القطب ويفضل أن يكون العنصر الأوسط في المصفوفة.
2. ضع القطب في مكانه الصحيح بحيث تكون كل العناصر الأقل منه على يساره، وكل العناصر الأكبر منه على يمينه.
3. ينتج بناءً على الخطوة السابقة قائمتان جزئيتان الأولى على يسار القطب، والأخرى على يمينه.
4. إذا احتوت القائمة اليسرى على أكثر من عنصر أعد تطبيق الخوارزمية على هذه القائمة.
5. بالمثل إذا احتوت القائمة اليمنى على أكثر من عنصر أعد تطبيق الخوارزمية عليها.



مثال (4)

افرض أنك ترغب في ترتيب عناصر المصفوفة التالية ترتيباً تصاعدياً باستخدام الفرز السريع:

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
8	22	30	16	20	18	12	14	25

ففي الخطوة الأولى يتم اختيار عنصر معين من عناصر المصفوفة حيث يتم تقسيم العناصر المتبقية إلى مجموعتين، تحتوي المجموعة الأولى على جميع العناصر الأقل قيمة من العنصر الذي تم اختياره. أما قيم عناصر المجموعة الثانية فهي أكبر من قيمة ذلك العنصر.

ويوجد أمامك عدة خيارات لاختيار العنصر، كأن يكون العنصر الأول، أو العنصر الأخير

أو العنصر الأوسط. ولكن أثبتت الدراسات أنه كلما كان العنصر الذي وقع عليه الاختيار قريباً من الوسيط للمصفوفة،(القيمة التي تتوسط مجموعة العناصر) كلما زادت كفاءة الفرز السريع.

وفي هذا المثال، فإن القيمة التي تتوسط مجموعة العناصر A[4] تمثل بالعنصر وقيمته 20 . لذلك سيتم اختيار هذه القيمة كخطوة أولى في الفرز السريع.

أما الخطوة التالية فتتمثل في تعيين مؤشرين (I وJ) للتأشير على عناصر المجموعة، بحيث يستعمل المؤشر I للتأشير على بداية القائمة (أي أن القيمة المبدئية للمؤشر I تساوي 0) ويستعمل المؤشر J للتأشير على نهاية المصفوفة (أي أن القيمة المبدئية للمؤشر J تساوي 8 لكونه يساوي عدد عناصر المصفوفة).

ويمكن توضيح ذلك على النحو التالي:

8	22	30	16	**[20]**	18	12	14	25
↑								↑
I= 0								J= 8

ويلاحظ أن القيمة 20 هي القيمة التي تم اختيارها كقطب لتوسطها بين قيم المصفوفة.

وفي الخطوة التالية تبدأ عملية مقارنة قيمة عنصر المصفوفة المشار إليه بالمؤشر I مع القطب، والهدف هو إيجاد العناصر الأكبر من القطب، وذلك لنقلها إلى يمينه، وفي كل مقارنة إذا كانت قيمة العنصر أقل من القيمة 20 (القطب)، يتم تعديل قيمة المؤشر I ليشير إلى العنصر التالي (أي أنه يتم زيادة القيمة 1 إلى قيمة I)، ويترك العنصر في مكانه. وتتوقف عملية المقارنة عند الوصول إلى عنصر قيمته أكبر من القيمة 20. وهذا يعني أن مثل هذه القيمة يجب أن تكون إلى يمين القطب (أي ضمن العناصر التي تزيد قيمتها عن القطب). وفي مثالنا تستمر المقارنة إلى أن نصل إلى العنصر 22 وهو أكبر من القطب (20).

ومن الجهة الأخرى للمصفوفة، تبدأ عملية المقارنة بين العنصر المشار إليه بالمؤشر J مع القطب، والهدف هو إيجاد العناصر الأقل من القطب، وذلك لنقلها إلى يساره. وطالما أن المؤشر J يشير إلى عنصر أكبر قيمة من 20 فإنه يتم تنقيص القيمة 1 من قيمة المؤشر J وذلك للانتقال إلى العنصر السابق في هذا الجزء من العناصر.

وبالرجوع إلى المصفوفة تلاحظ أن 25 أكبر من 20؛ لذا يجب أن تبقى في محلها،

ثم تنقص قيمة J بمقدار واحد فنصل بذلك إلى القيمة 14، وبما أن القيمة 14 أصغر من القيمة 20 لذلك تتوقف عملية التنقيص من قيمة المؤشر J ونحصل على الترتيب التالي:

8	22	30	16	[20]	18	12	14	25
	↑							↑
	I= 1							J= 8

ولعلك تلاحظ، عزيزي الدارس، أنه تم حتى هذه اللحظة إيجاد عنصرين يجب استبدالهما مع بعضهما البعض وذلك لكي يصبح كل منهما في الجهة الصحيحة بالنسبة إلى القيمة 20 في القائمة المرتبة وذلك لكون القيمة 14 أصغر من القيمة 20 (أي ينبغي أن تقع قبلها في القائمة المرتبة) ولكون القيمة 22 أكبر من 20 (أي يجب أن تقع إلى يمين القيمة 20 في القائمة المرتبة ترتيباً تصاعدياً).

وفي كل مرة يتم فيها تبديل عنصرين مع بعضهما البعض يجب إضافة القيمة 1 إلى المؤشر I، وطرح القيمة 1 من المؤشر J. ولذلك نحصل بعد هذه الخطوة على الترتيب التالي لأرقام المصفوفة:

8	22	30	16	[20]	18	12	14	25
		↑		↑				
		I= 2		J= 6				

ونقوم من ثم بتكرار الخطوة السابقة، فطالما أن المؤشر I يشير إلى عنصر قيمته أصغر من 20 يجب زيادة قيمة I بمقدار 1. وكذا الأمر من الجهة الأخرى للمصفوفة (أي جهة J)، فطالما أن المؤشر J يشير إلى عنصر قيمته أكبر من 20 يجب تنقيص قيمة المؤشر J بمقدار 1، وإعادة مقارنة العنصر الجديد مع القيمة 20 إلى حين مقابلة عنصر قيمته أقل من 20، فعندها يتم تبديل العنصر الذي يشير إليه المؤشر I مع العنصر الذي يشير إليه المؤشر J.

وبما أن القيمة 30 أكبر من 20 وكذلك القيمة 12 أصغر من القيمة 20، يجب تبديل هاتين القيمتين مع بعضهما البعض ومن ثم زيادة I بمقدار 1 وتنقيص J بمقدار 1، كما هو مبين:

8	22	30	16	[20]	18	12	14	25
			↑		↑			
			I= 3		J= 5			

وهكذا نكرر العملية مرة أخرى ونقارن القيمة 16 مع القيمة 20. وبما أن 16 أقل من 20 تعمل على زيادة المؤشر I بمقدار 1 (أي أن قيمة I تصبح 4) بحيث يصبح المؤشر I يشير إلى القيمة 20، وتتم بعد ذلك مقارنة القيمة التي يشير إليها المؤشر I (وهي 20) مع

20. وبما أن 20 ليست أقل من 20 لا تطرأ أي زيادة على قيمة I، وتتوقف عملية المقارنة بين القيم المؤشر إليها بالمؤشر I والقيمة 20.

ثم ننتقل إلى الجهة الأخرى من المصفوفة، أي إلى جهة المؤشر J، فبما أن المؤشر J يشير إلى عنصر قيمته أكبر من 20، يجب تنقيص قيمة J بمقدار 1. وبما أن القيمة المشار إليها بالمؤشر J (أي القيمة 18) ليست أكبر من 20، لا يطرأ أي تنقيص على قيمة J. والآن تبدأ المبادلة بين العنصرين 18 و20، وتزداد قيمة I بمقدار 1 وكذلك تنقص قيمة المؤشر J بمقدار 1. ويمكن توضيح ما جرى إلى الآن على النحو التالي:

8	22	30	16	[20]	18	12	14	25
			↑ J=4	↑ I=5				

تلاحظ، عزيزي الدارس، الآن أن قيمة I أصبحت أكبر من قيمة J وهذا يعني انتهاء الجولة الأولى من هذه الخوارزمية، حيث تصبح المصفوفة الكلية مرتبة جزئياً ومقسمة إلى قسمين، يشمل القسم الأول جميع العناصر إلى يسار القيمة 20 وقيمة كل عنصر من هذه العناصر أقل من القيمة 20. وهذه العناصر هي:

8 14 12 16 18

أما القسم الثاني (المجموعة أو القائمة الثانية) فتشمل جميع العناصر من القيمة 20 وإلى نهاية المصفوفة. وهذه العناصر هي:

20 30 22 25

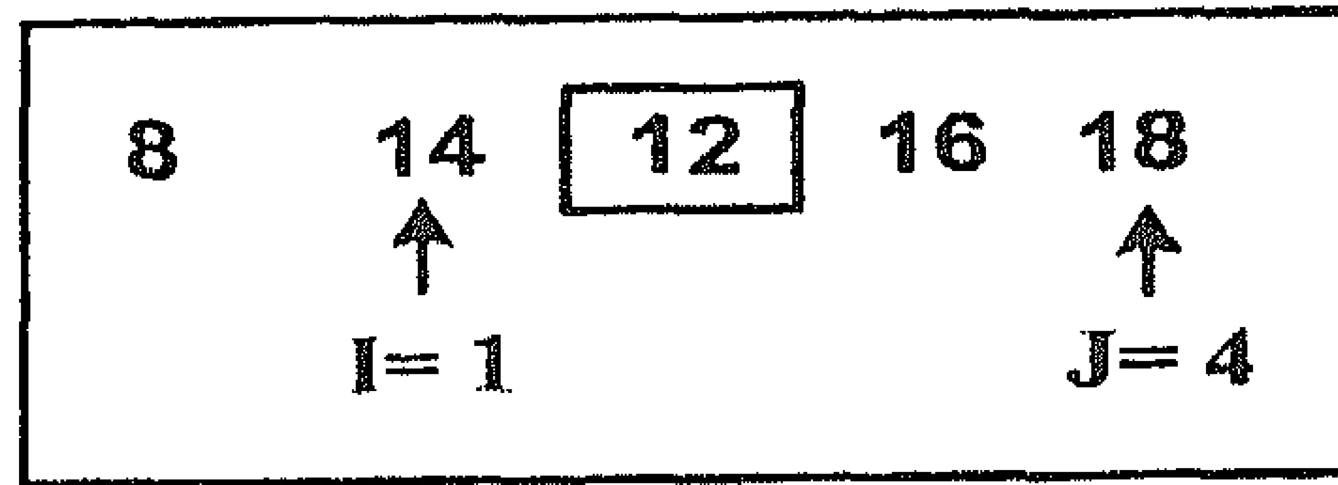
وهذا يعني أن القائمة اليسرى تتكون من العنصر الأول للمجموعة وحتى العنصر المشار إليه بالمؤشر J، وأن القائمة اليمنى تبدأ بالعنصر المشار إليه بالمؤشر I وحتى نهاية المجموعة.

وهذا ينطبق على جميع المجموعات عند تقسيمها إلى مجموعتين:

وبعد الانتهاء من المرحلة الأولى والتي كانت نتائجها قائمتين، كرر جميع الخطوات السابقة على القائمتين الجزئيتين الجديدتين. خذ القائمة الأولى على سبيل المثال، واختر الرقم (12) لتوسطه القائمة كقطب كما هو مبين في الترتيب التالي، ثم حاول وضع هذا القطب في مكانه الصحيح بحيث تكون كل العناصر الأقل منه على يساره، وكل العناصر الأكبر منه على يمينه.

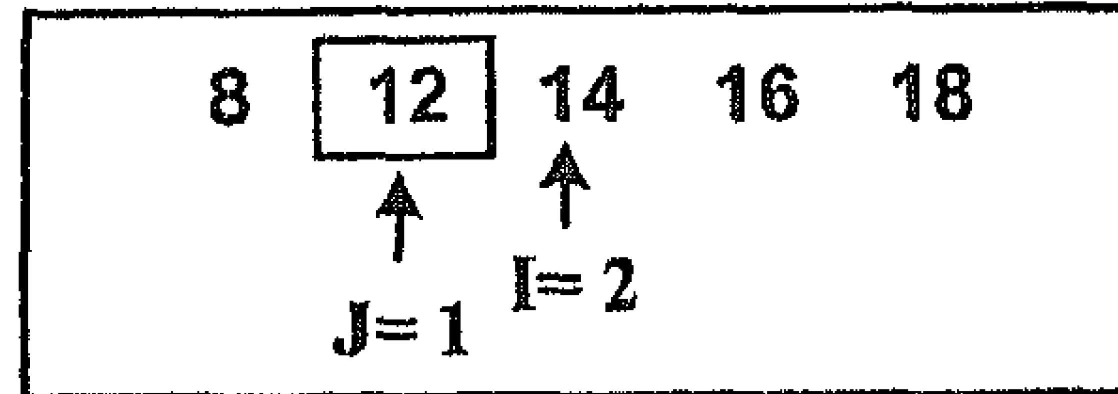
8	14	[12]	16	18
↑ I=0				↑ J=4

وبما أن (8) أقل من (12) تضيف إلى المؤشر (I) القيمة 1. فتصبح قيمة I تساوي 2 وتحصل على الترتيب التالي:



وبما أن المؤشر I أصبح يشير إلى أكبر من القيمة (12) تتوقف عملية زيادة قيمة المؤشر I وتنتقل المقارنة إلى طرف المؤشر J. وبما أن 18 أكبر من 12 يتم تنقيص قيمة J بمقدار 1. وتستمر المقارنة، وتقارن الآن 16 مع 12. وبما أن 16 أكبر من 12 تتم عملية تنقيص قيمة J مرة أخرى، فيصبح المؤشر J يشير إلى القيمة 12. وبما أن 12 ليست أكبر من 12 تتوقف عملية التعديل على قيمة المؤشر J. بعد ذلك تتم عملية تبديل محتويات الموقعين المشار إليهما بالمؤشر I والمؤشر J أحدهما بالآخر.

ومن ثم تضيف القيمة 1 إلى I وتطرح القيمة 1 من J، بحيث تحصل على الترتيب التالي:



وبما أن (J<I) فإنك تكون قد قسمت هذه القائمة إلى قائمتين أخريين: الأولى وتشمل العناصر 12 و8 أي من العنصر الأول إلى العنصر J، والثانية وتشمل العناصر 14, 16, 18 (أي من العنصر I إلى العنصر الأخير) وهذا يشكل نهاية جولة الترتيب الثانية بتقسيم القائمة اليسرى إلى قائمتين كما هو مبين.

وعند تطبيق دالة الفرز مرة أخرى على القائمة الأولى تقسم هذه القائمة إلى قائمتين تحتوي كل منهما على عنصر واحد كما يلي:



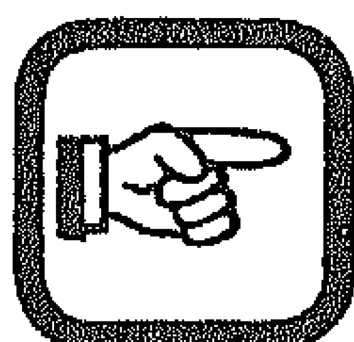
القائمة اليمنى	القائمة اليسرى
12	8

وتلاحظ هنا أن القائمة اليسرى تتكون من عنصر واحد. ولذلك لا داعي لتنفيذ دالة الفرز عليها.

ويمكنك ملاحظة ذلك بشكل عام على أي قائمة جزئية من القوائم اليسرى، حيث أن العناصر المكونة لأي قائمة من هذه القوائم يشار إليها بالمؤشرات First إلى J. وعندما تكون قيمة J مساوية لقيمة First (أي تشير إلى أول عنصر في القائمة) فهذا يعني أن القائمة تحتوي على عنصر واحد ولا داعي لإجراء عملية الفرز على هذا العنصر.

وعند إنهاء عملية الفرز على قائمة من القوائم، تنتقل الخوارزمية إلى القائمة التي تليها. وبتنفيذ الخطوات السابقة نفسها على كل هذه القوائم، تحصل في النهاية على القائمة الكلية مرتبة ترتيباً تصاعدياً حسب الأصول.

لاحظ، عزيزي الدارس، أن عملية تقسيم قائمة من القوائم إلى قائمتين قد تعطي قائمتين لا تحوي أي منهما على العنصر الذي تم اختياره (Pivot) عند تطبيق دالة الفرز وذلك كما هو مبين في المثال التالي:



مثال (5)

افرض على سبيل المثال أنك أعطيت الأعداد التالية:

5 10 15 20 25 30 40

وإنك ترغب في تطبيق الفرز السريع عليها لترتيبها ترتيباً تصاعدياً، (لاحظ أولاً أن القائمة مرتبة أصلاً).

افرض أنك اخترت العدد 20 كمحور (Pivot).

عند تطبيق الجولة الأولى من الفرز السريع تحصل على الترتيب التالي:

5	10	15	**[20]**	25	30	40
			↑ I= J=1			

والآن يجب تبديل العنصرين المشار إليهما بالمؤشرين I وJ مع بعضهما البعض، وبما أن I يساوي J فلا يحصل أي تغيير في القيم.

بعد ذلك يتم زيادة القيمة 1 إلى I وتنقيص القيمة 1 من J فنحصل على الترتيب التالي:

5	10	15	**[20]**	25	30	40
		↑ I= 2		↑ J= 4		

وبما أن قيمة I أصبحت أكبر من قيمة J تنتهي هذه الجولة. وهكذا تكون القيمة 20 خارج القائمتين الجزئيتين الجديدتين.

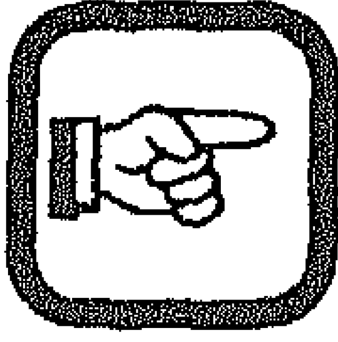
تتكون القائمة الجزئية الأولى من العناصر:

15 10 15

والقائمة الجزئية الثانية من العناصر:

25 30 40

ولفهم دالة الفرز السريع حاول حل المثال التالي بمفردك ثم قارن الحلين.



مثال (6)

افرض أنك ترغب في تنفيذ الفرز السريع على مجموعة العناصر التالية وذلك لترتيبها ترتيباً تصاعدياً بحيث يتم اعتبار العنصر الأول في كل قائمة من القوائم على أنه العنصر المحور (Pivot) عند تطبيق كل جولة من جولات الفرز السريع. والمطلوب هو بيان حالة المصفوفة في كل جولة من الجولات:

[20 10 35 2 60 13 55 15 45 18]

الحل:

في الجولة الأولى يتم اختيار العنصر الأول وقيمته 20 على أنه المحور بحيث يتم تقسيم العناصر إلى مجموعتين، تحتوي المجموعة الأولى على جميع العناصر التي قيمها أقل من القيمة 20، وتحتوي المجموعة الثانية على العناصر المتبقية. وللتوضيح يمكن وضع عناصر كل مجموعة من المجموعات داخل قوسين مربعين، ويوضع العنصر الذي تم اختياره كمحور في محله الترتيبي. وبعد نهاية الجولة الأولى تحصل على الترتيب التالي:

[13 10 18 2 15] <u>20</u> [55 60 45 35]

وقد حصلنا على ذلك بأن أخذنا القيمة 20 وبدأنا بالمؤشرين I وJ حيث يشير I إلى القيمة 10 (أي العنصر الثاني من المجموعة الأولى) وتشير J إلى العنصر الأخير من المجموعة الثانية أي القيمة 18. وبما أن 10 أقل من 20 تعدل قيمة العداد I بحيث يشير إلى القيمة 35. ولذلك يتم تبديل العنصر الثالث مع العنصر الأخير وذلك لكون القيمة 35 أكبر من 20 والقيمة 18 أقل من 20. ومن ثم يتم تعديل قيم المؤشرين I وJ ويشير كل منهما إلى العنصر التالي في مجموعته.

وتصبح محتويات المصفوفة كما يلي:

20	10	18	2	60	13	55	15	45	35
			↑ I					↑ J	

بعد ذلك يتم مقارنة 2 مع 20، وبما أن 18 أقل من 2 ينتقل المؤشر I إلى القيمة التالية وهي 60 (وهذه القيمة أكبر من 20). وكذلك فإن المؤشر J ينتقل إلى القيمة 15 ومن ثم يتم تبديل محتويات العنصرين الخامس والثامن وتصبح محتويات المصفوفة كما يلي:

20	10	18	2	60	13	55	15	45	35
					↑ I	↑ J			

بعد ذلك يتم مقارنة القيمة 13 مع 20 وتعدل قيمة I بحيث يشير إلى 55. وكذلك تعدل قيمة J، بحيث يشير إلى القيمة 13. وتصبح محتويات المصفوفة كما يلي مع قيم المؤشرين I وJ:

20	10	18	2	60	13	55	15	45	35
					↑ J	↑ I			

وبما أن قيمة I أكبر من J تنتهي هذه الجولة وذلك بتبديل العنصر الأول من المجموعة مع العنصر13، وتصبح المصفوفة مقسومة إلى قسمين القسم الأول يحتوي على القيم أقل من 20، والقسم الثاني يحتوي على القيم أكبر من 20، ولذلك فإن القيمة 20 تصبح في مكانها الصحيح بالنسبة لجميع القيم في المصفوفة. وتصبح المصفوفة مع نهاية هذه الجولة على النحو التالي:

[13 10 18 2 15] <u>20</u> [55 60 45 35]

وفي الجولة الثانية نكرر العملية على القائمة اليسرى، ونعتبر القيمة 13 على أنها العنصر المحور. وفي نهاية الجولة تحصل على الترتيب التالي للمصفوفة:

2 10 <u>13</u> [18 15] <u>20</u> [55 60 45 35]

وفي الجولة التالية تحصل على المصفوفة التالية:

حيث أن العناصر الثلاثة الأولى تصبح مرتبة وفق الترتيب التالي:

2 10 13 [18 15] 20 [55 60 45 35]

وفي الجولات المتبقية تحصل على المصفوفة مرتبة كما هو مبين:

2 10 13 15 18 20 [55 60 45 35]

2 10 13 15 18 20 [45 35] 55 [60]

2 10 13 15 18 20 35 45 55 [60]

وأخيراً نحصل على القائمة النهائية مرتبة ترتيباً تصاعدياً كما يلي:

2	10	13	15	18	20	35	45	55	60

لا بد أنك لاحظت، عزيزي الدارس، أنّ من الممكن استخدام الاستدعاء الذاتي (Recursion) لتنفيذ خوارزمية الفرز السريع، وذلك لأن الخاصيتين اللتين تجعلان أي خوارزمية مناسبة للاستدعاء الذاتي متحققتان في هذه الخوارزمية وهما:

1. وجود الحالة الأساس التي لا يحتاج حلها إلى الاستدعاء الذاتي. وهذه الخاصية متحققة لذا لا نحتاج لفرز القائمة التي تحتوي على عنصر واحد، أو أقل لاستخدام الاستدعاء الذاتي.
2. في كل مرة تستدعى الخوارزمية استدعاءً ذاتياً تستدعى حالة أقرب إلى الحالة الأساس. وهذه الخاصية أيضاً متحققة إذ تستدعى الخوارزمية في كل مرة على قائمة جزئية أصغر من القائمة الكلية، وفي نهاية الأمر لا بد وأن يصبح حجم القائمة عنصراً واحداً، أو أقل أي نصل إلى الحالة الأساس.

وفيما يلي دالة الفرز السريع باستخدام الاستدعاء الذاتي مكتوباً بلغة البرمجة سي++، ويعكس هذه الدالة طبيعة هذا النوع من أنواع الفرز:

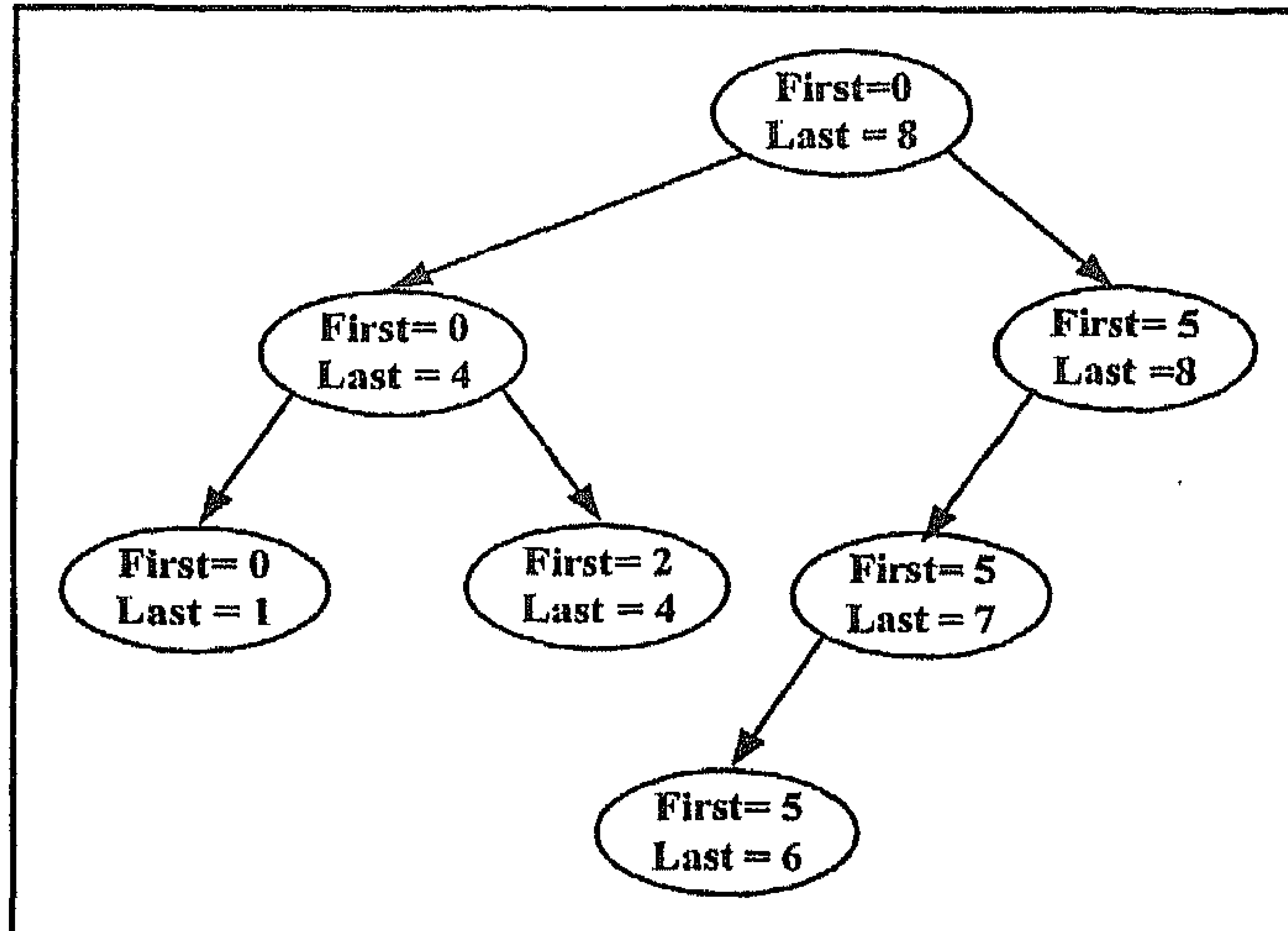
```
//QSort-Control function  Parmeters  A: Array of integer numbers
// Value parameters N:  Number of elements in the array.
void QSort(ArrayType& A,int First,int Last)
 {int I,J;   //Array indices
  int Pivot; //Middle array value
  int temp;  //Temporary variable
  I=First;
  J=Last;    //Pivot
  Pivot=A[(First+Last)/2];   //Find middle array value
                              ضع القطب في مكانه الصحيح//
  do
   {while (A[I]<Pivot) I++;
    while (A[J]>Pivot) J--;
    if (I<=J)
     { //Switch A [I] with A [J]
      temp=A[I];
      A[I]=A[J];
      A[J]=temp;
                    I++;
      J--;
     }
   }
  while (I>J);
  إذا احتوت القائمة اليسرى على أكثر من عنصر استدع  QSort لترتيبها ذاتياً //
  if (First<J)
   QSort(A,First,J);
```

```
//Make recursive call to QSort with same First
   //value and Last value set to J
 // إذا احتوت القائمة اليمنى على أكثر من عنصر استدع QSort لترتيبها ذاتياً
 if (I<Last)
   QSort(A,I,Last);
 }
```

تحليل خوارزمية الفرز السريع

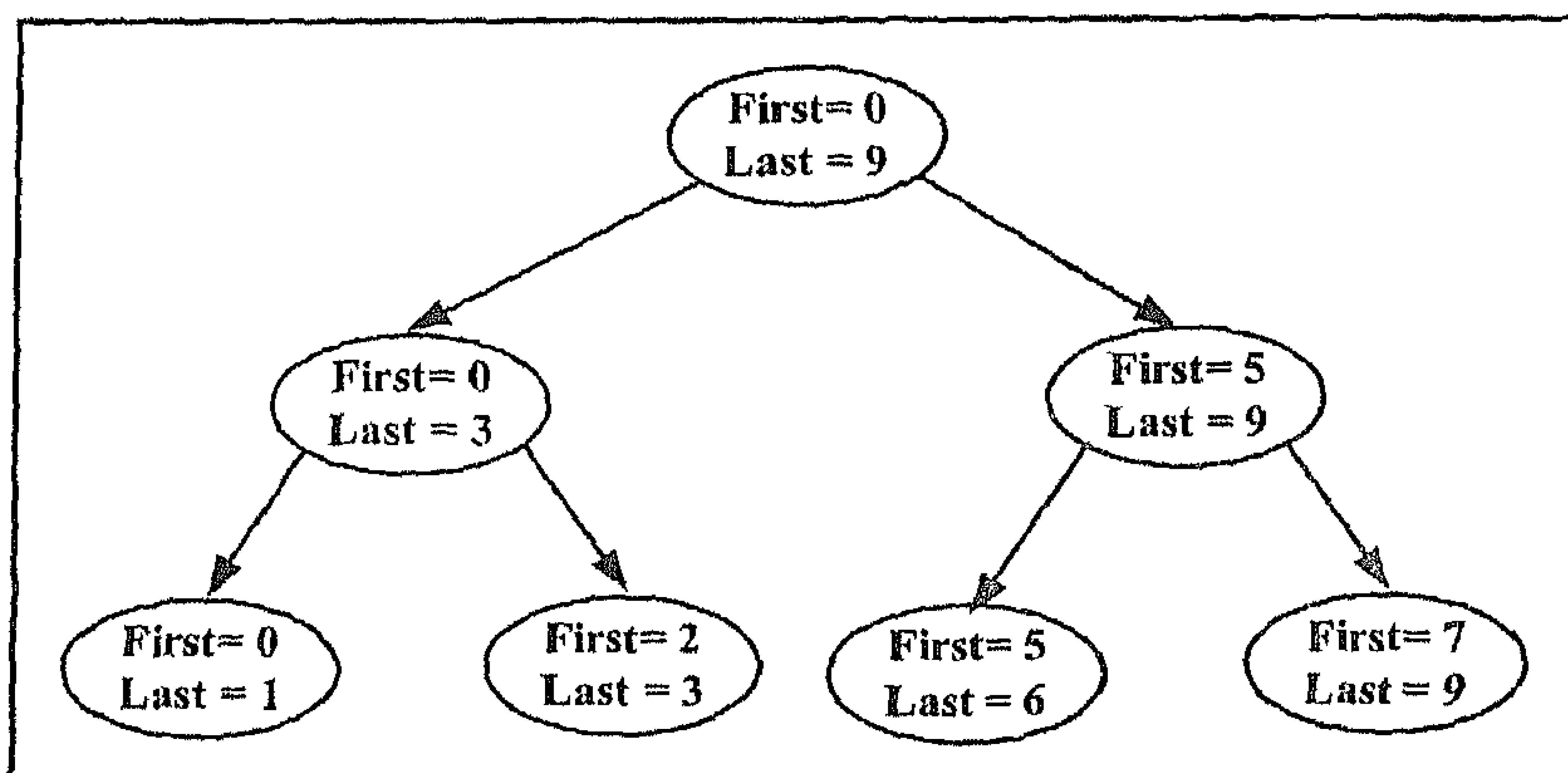
لتحليل خوارزميات الاستدعاء الذاتي يلزم عادة رسم ما يسمى بشجرة الاستدعاء الذاتي Tree of Recursive Calls، وهي تبين عدد المرات التي تستدعى بها الخوارزمية ذاتياً وقيم عواملها (Parameters) حيث يمثل كل عنصر (Node) في الشجرة عملية استدعاء ذاتي. وتعتمد كفاءة الخوارزمية على حجم هذه الشجرة، وعلى حجم العمل الذي يتطلبه كل عنصر في الشجرة (كل استدعاء). ويمثل الشكل (3) مثالاً على شجرة الاستدعاء الذاتي للدالة QSort عند عملها مع المصفوفة الواردة في المثال (4).

بما أن كل عنصر في شجرة الاستدعاء يمثل استدعاءً للخوارزمية، فإن كفاءتها تعتمد على عدد العناصر في الشجرة وعلى تكلفة تنفيذ كل عنصر (استدعاء). ولخوارزمية الفرز السريع حالة مثلى (الأفضل) Best Case حيث تأخذ أقل وقت ممكن، وحالة سيئة جداً Worst Case تأخذ وقتاً أكبر بكثير. ويمثل الشكل (4) شجرة الاستدعاء الذاتي للحالة الأفضل. إن الحالة الأفضل لخوارزمية الفرز السريع هي الحالة التي يوضع فيها القطب (Pivot) في منتصف القائمة، وعليه يكون عمق شجرة الاستدعاء متناسباً مع $\log_2 n$.



الشكل (3): Tree of Recursive Call

شجرة الاستدعاء الذاتي لـ Qsort عند استدعائها على المصفوفة الواردة في المثال (4)



شكل (4): يمثل الحالة الأفضل لخوارزمية الفرز السريع عند استدعائها على مصفوفة من 10 عناصر

الآن عرفنا حجم الشجرة الأفضل، ولكن ما هي تكلفة كل عنصر في هذه الشجرة؟ أي ما عدد الخطوات التي ستنفذ في كل استدعاء؟

لنأخذ على سبيل المثال جذر الشجرة عندما First = 0 و Last = 9 إن ما تعمله Qsort هو اختيار القطب، ثم وضعه في المكان المناسب (من خلال دوران do-while) حيث تبدأ قيمة J عند n وقيمة I عند واحد. ويستمر الدوران إلى أن تصبح قيمة J أقل من I، عندها نكون قد نفذنا جملة if داخل الدوران n من المرات. وبنفس الطريقة عند العنصر الأول في المستوى الثاني في شجرة الاستدعاء (أي عند First = 0 وLast = 3) يتضح أن عدد المقارنات سيكون n/2 ولكن لاحظ، عزيزي الدارس، أن هنالك عنصراً آخر على نفس المستوى (الثاني) سيأخذ أيضاً n/2 من المقارنات مما يعني أن مجموع المقارنات (if statement) التي ستنفذ في هذا المستوى هي n من المقارنات.

وبطريقة مشابهة تستطيع استنتاج أن كل عنصر في المستوى الثالث سيحتاج n/4 من المقارنات، وبما أن هنالك 4 عناصر، فإن هذا المستوى يحتاج أيضاً إلى n من المقارنات.

وبما أن لدينا $\log^2 n$ من المستويات (عمق الشجرة)، وكل مستوى سيكلف n من المقارنات، فإننا نستطيع استنتاج أن كفاءة الخوارزمية هي $O(n \log^2 n)$.

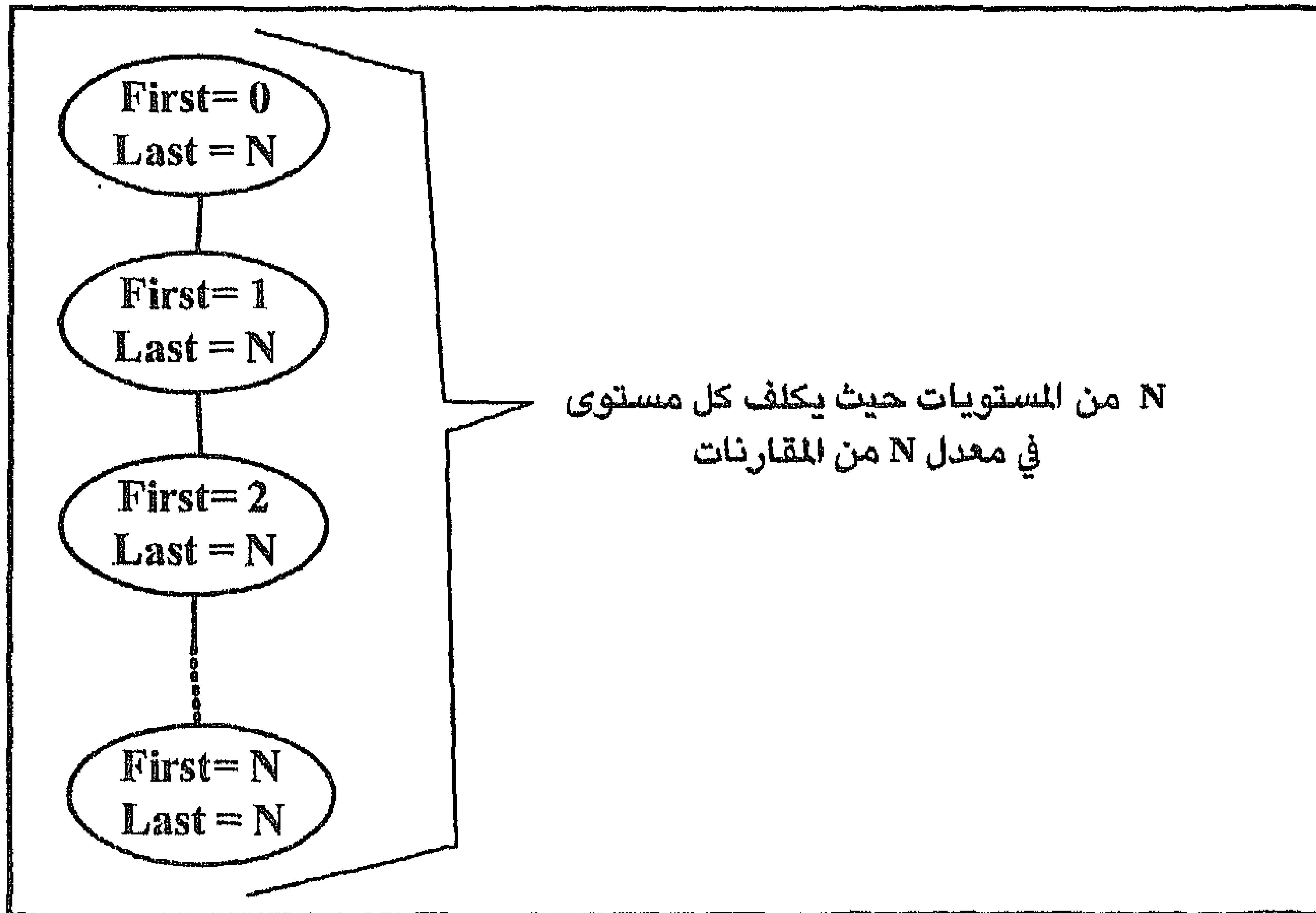
إن الحالة الأسوأ لخوارزمية الفرز السريع هي الحالة التي لا يتم فيها وضع القطب في منتصف القائمة، بل في بدايتها، وبذلك فإن القطب لا يقسم القائمة إلى قائمتين متساويتين في الطول، بل إلى قائمة تتكون من عنصر واحد، وأخرى من n-1 من العناصر، وهكذا.

وبذلك نحصل على شجرة استدعاء طويلة جداً عمقها n من المستويات كما يظهر في الشكل (5).

لاحظ أن المستوى الأول سيكلف n من المقارنات، والمستوى الثاني n – 1 من المقارنات والثالث 2 - n وهكذا سيكلف الأخير مقارنة واحدة، وبجمع هذه الأعداد معاً يظهر عدد المقارنات الكلي.

عدد المقارنات الكلي في الحالة الأسوأ = $n^2 / 2 = \frac{(n - 1 + 1)n}{2}$

وعليه فإن خوارزمية الفرز السريع في الحالة الأسوأ تكون تون $O(n^2)$



شكل (5): يبين الحالة الأسوء لخوارزمية الفرز السريع

3.1.3 الفرز المقيد (Heap Sort)

سمي هذا الفرز بهذا الاسم لاستخدامه على البيانات الممثلة على شكل شجيرة ثنائية ذات خواص معينة (مقيدة). لذلك قبل إجراء هذا الفرز، لا بد من تحويل العناصر المراد فرزها، والتي قد تكون ممثلة على شكل مصفوفة إلى شجيرة ثنائية مقيدة يطلق عليها اسم الشجيرة الثنائية المقيدة (Heap)، والتي تتميز بالخواص التالية:

1. يجب أن تكون الشجيرة ثنائية كاملة.
2. يجب أن تكون القيمة المخزنة في كل موقع (Node) من مواقع الشجيرة الثنائية

الكاملة أصغر من القيمة المخزنة في الموقع الذي يشكل الأهل (Parent) لهذا الموقع أو مساوية لها.



مثال (7)

على سبيل المثال افرض أنك أعطيت مصفوفة مكونة من عشرة عناصر من النوع العددي الصحيح كما هو مبين:

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
16	19	14	10	15	7	20	8	12	18

وأنك ترغب في تحويل هذه العناصر إلى شجيرة ثنائية كاملة مقيدة وذلك بإعادة ترتيب عناصر المصفوفة بحيث ينطبق عليها التعريف السابق (أي تعريف الشجيرة الثنائية المقيدة).

كما تعلم، لو فرضنا أن عنصراً ما يقع في الموقع i فإن الابن الأيسر لهذا الموقع (إن وجد) سوف يقع في الموقع 2i وكذلك فإن الابن الأيمن (إن وجد) سوف يقع في الموقع (2i + 1)، وهذا نابع من تعريف الشجيرة الثنائية الكاملة عند تمثيلها في مصفوفة كما هو في المثال الحالي.

وتتم عملية إنشاء الشجيرة الثنائية المقيدة (Heap) داخل المصفوفة وذلك حسب الخطوات التالية:

1. خذ العنصر الأول والذي يحتوي القيمة 16 واعتبره شجيرة ثنائية مقيدة تحتوي على عنصر واحد قيمته (16).
2. كون شجيرة ثنائية مقيدة من العنصرين الأول والثاني في المصفوفة (أي القيمتين 16 و19).
3. كون شجيرة ثنائية مقيدة من العناصر الثلاثة الأولى في المصفوفة وهي التي تحتوي على القيم (16, 19, 14).
4. كرر هذه العملية وذلك بإضافة عنصر جديد إلى الشجيرة الثنائية المقيدة في كل خطوة، حتى تحصل على شجيرة ثنائية مقيدة تحتوي على جميع العناصر المطلوب إجراء الفرز عليها.

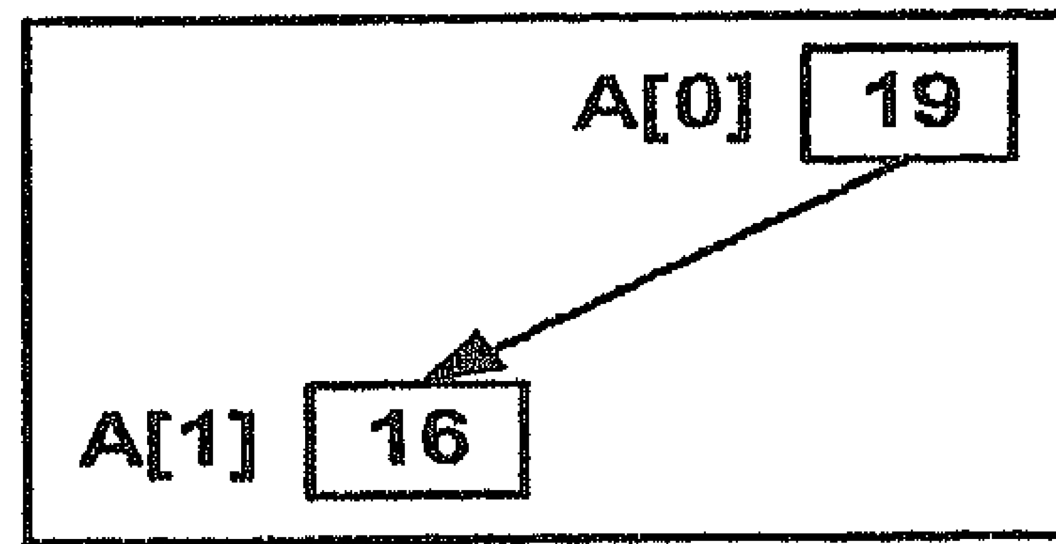
ويمكن بيان كيفية إجراء الخطوات المذكورة أعلاه (أي إنشاء شجيرة ثنائية مقيدة) كما يلي:

في البداية، يتم، عزيزي الدارس، إنشاء شجيرة ثنائية مقيدة تحتوي على عنصر واحد هو A[0] وقيمة هذا العنصر هي 16 كما هو مبين في المثال (7). ويعتبر هذا العنصر الجذر لهذه الشجيرة.

وفي الخطوة التالية (أي الخطوة الثانية) نأخذ العنصرين الأول والثاني (أي A[0]=16 وA[1]=19) ونكوّن شجيرة ثنائية مقيدة. ولكي نحافظ على خواص الشجيرة الثنائية المقيدة (أي أن قيم الأبناء أصغر من أو مساوية لقيمة الأهل) وبما أن A[1] هو الابن الوحيد لهذه الشجيرة يجب تبديل هذين العنصرين مع بعضهما البعض وتحصل على الشجيرة الثنائية المقيدة بعنصرين كما يلي:

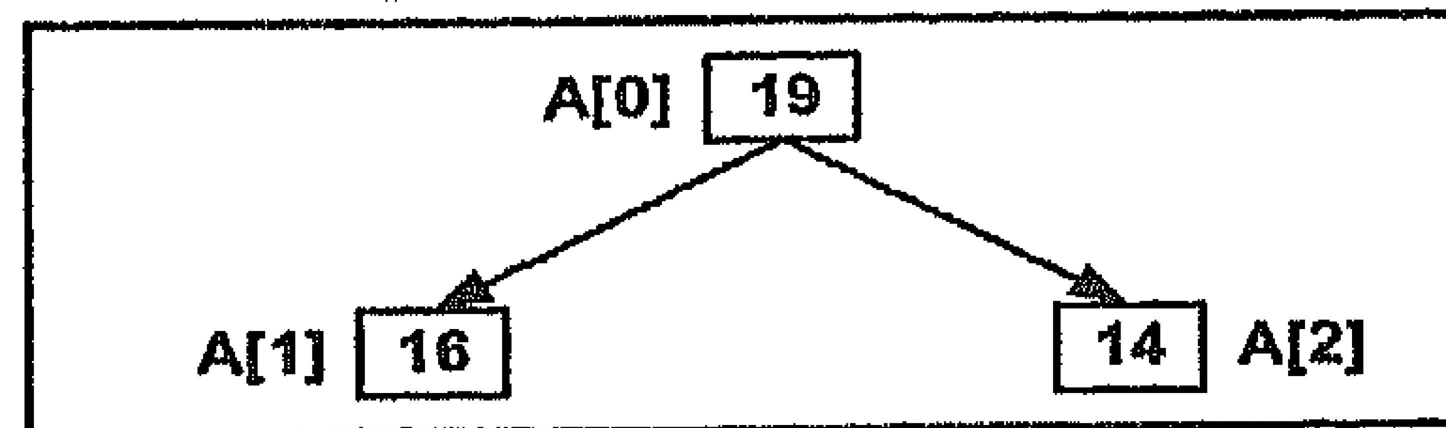
A[1]=16 A[0]=19

ويظهران على شكل هيكل شجري ثنائي مقيد كما يلي:

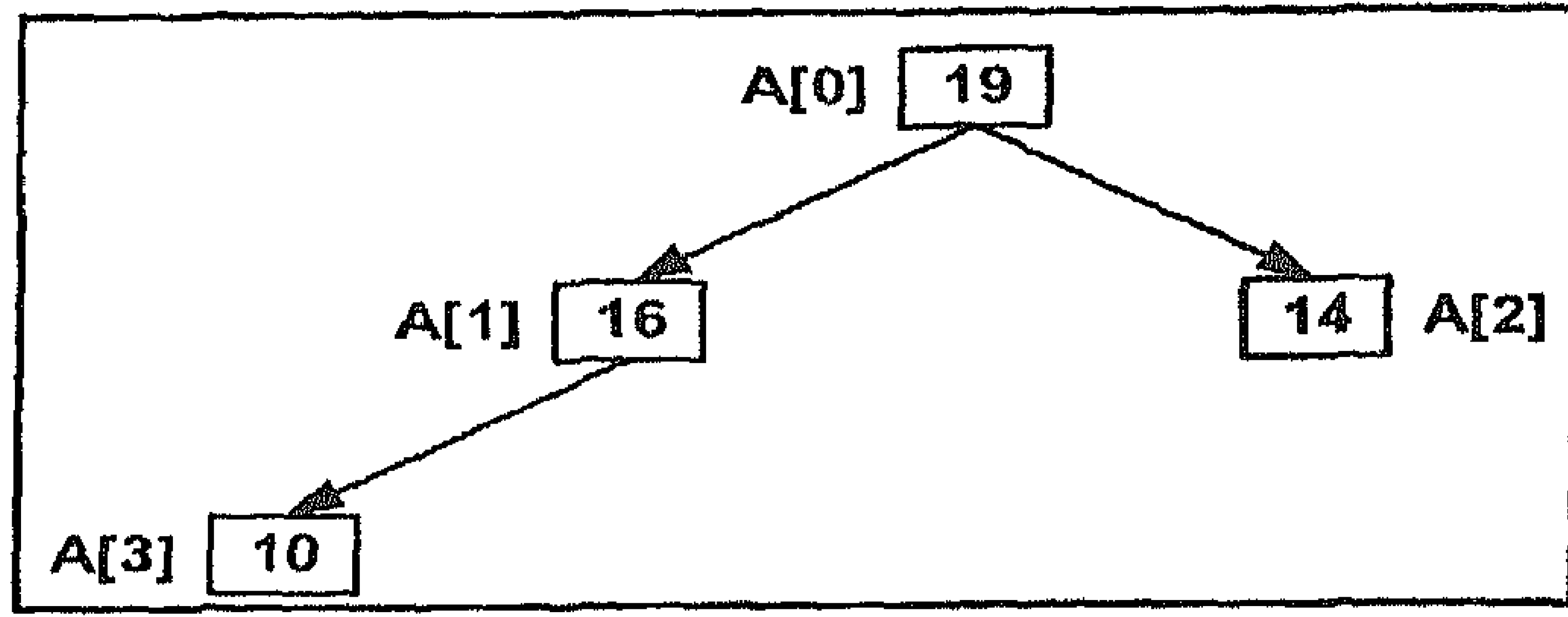


وفي الخطوة الثالثة خذ العنصر الثالث (A[2]=14) وكون شجيرة ثنائية مقيدة مكونة من العناصر الثلاثة الأولى.

لاحظ أن العنصر A[0] هو الأهل للعنصرين A[1] وA[2]. والعنصر A[1]هو الابن الأيسر للعنصر A[0] والعنصر A[2] هو الابن الأيمن للعنصر A[0]. وبما أن (A[2]=14) أصغر من A[0] الجديدة (أي أن محتويات الابن الأيمن أصغر من محتويات الأهل، يتبين أن العناصر الثلاثة A[0] وA[1] وA[2] تشكل شجيرة ثنائية مقيدة. ومن ثم لا حاجة لإجراء أي تعديل على محتويات عناصرها كما هو مبين في الترتيب التالي:

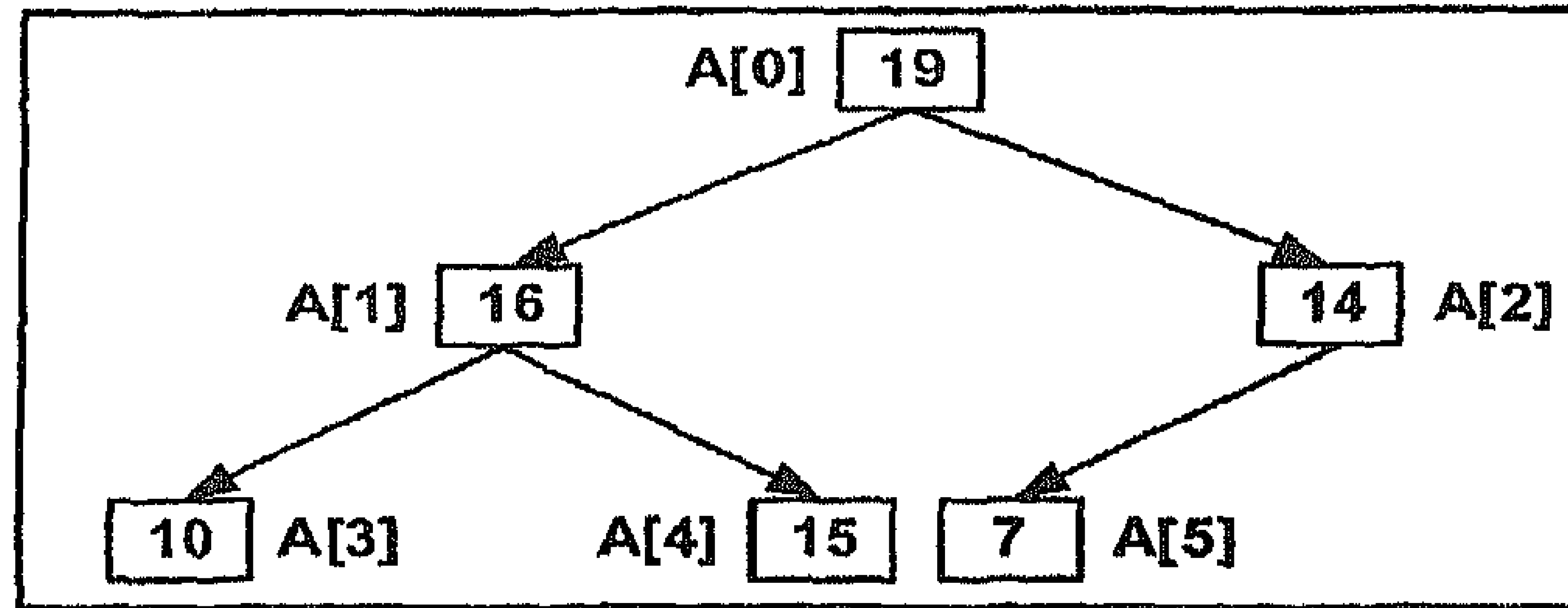


وفي الخطوة الرابعة نضيف إلى القائمة عنصراً جديداً (A[3] = 10) ونجري التعديلات اللازمة للحصول على شجيرة ثنائية مقيدة مكونة من أربعة عناصر. ويمكن تمثيل البيانات الموجودة في العناصر الأربعة على شكل شجيرة ثنائية كما يلي:



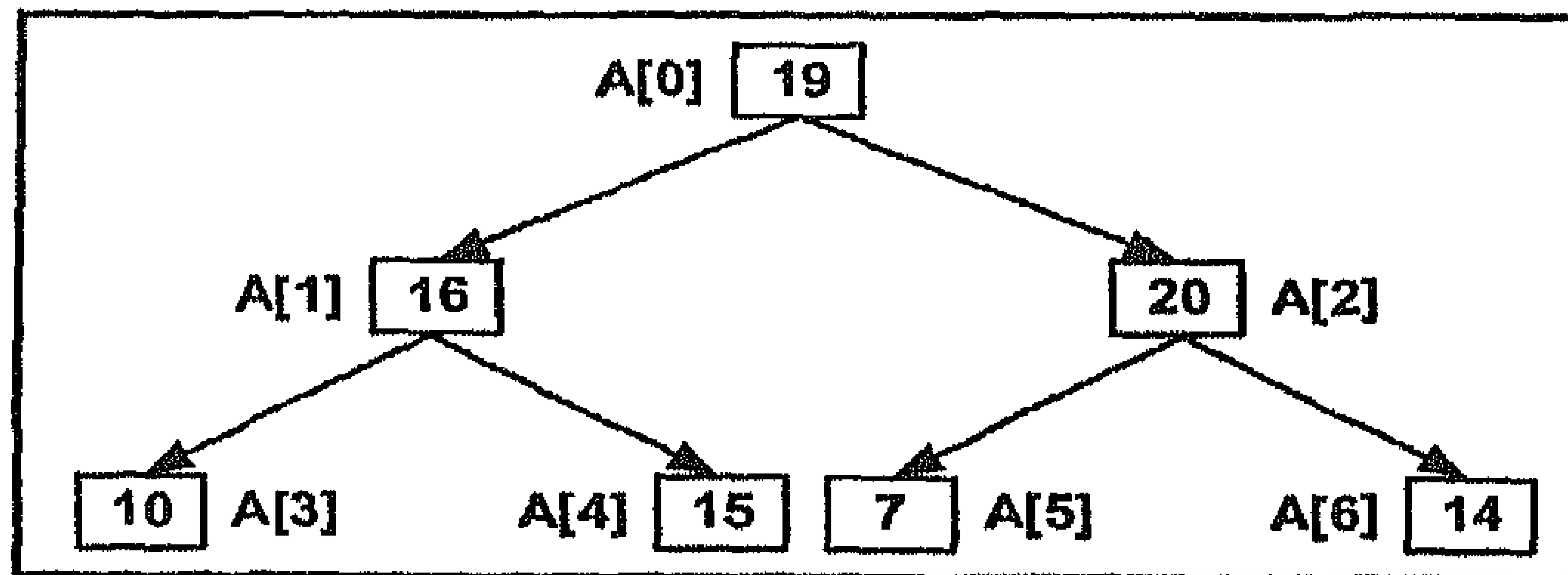
ونلاحظ على أن القيمة المخزنة في A[3] (أي الابن الأيسر لـ A[1]) أصغر من القيمة المخزنة في A[1] ولذلك فإن هذه العناصر الأربعة تشكل شجيرة ثنائية مقيدة. ولا حاجة لإجراء أي تعديل على عناصرها.

وتكرر العملية السابقة على العنصر الخامس A[4] وتجد أن العناصر ما زالت تشكل شجيرة ثنائية مقيدة، وكذلك الحال عند اعتبار العنصر السادس. ويمكن بيان ذلك بالترتيب التالي:

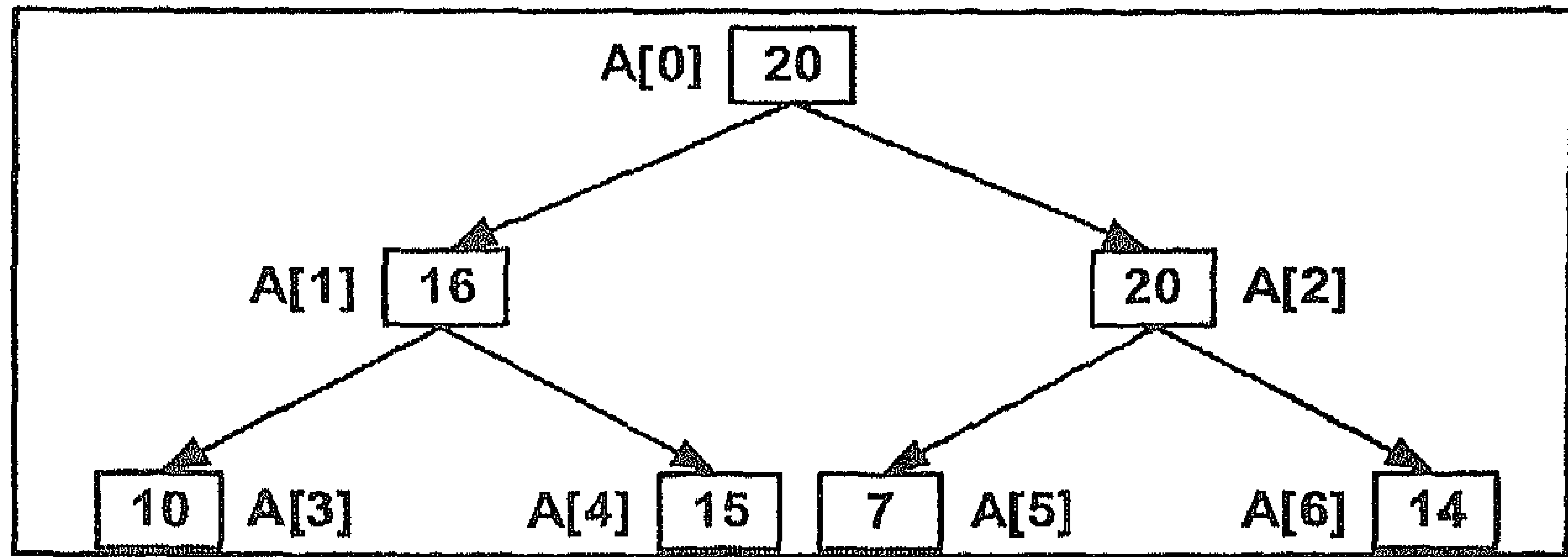


وفي الخطوة التالية وعند أخذ العنصر السابع (A[6] = 20) في المصفوفة بعين الاعتبار تجد أن هذا العنصر هو الابن الأيمن للعنصر (A[2]= 14). ولذلك فإن هذه العناصر لا تشكل شجيرة ثنائية مقيدة، وبالتالي لا بد من إجراء التعديل اللازم كما يلي:

أولاً: يجب تبديل العنصرين A[2] وA[6] مع بعضهما البعض لنحصل على الشكل التالي:

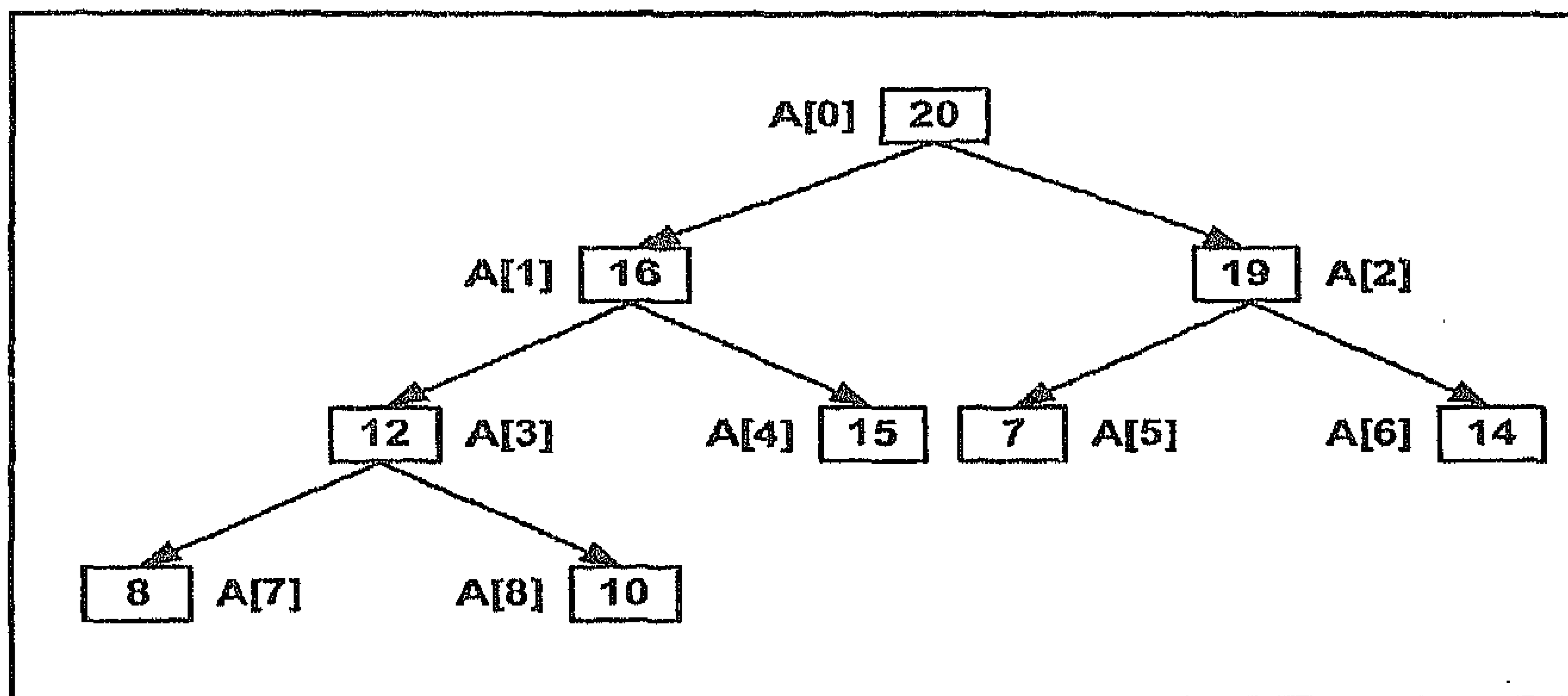


ثانياً: نلاحظ من التبديل أن قيمة A[2] (أي الابن الأيمن للعنصر A[0]) أصبحت أكبر من قيمة A[0]، ولذلك لا بد من تبديل محتويات العنصرين A[2] وA[0] لتحصل على شجيرة ثنائية مقيدة مكونة من سبعة عناصر A[0] إلى A[6] كما هو مبين أدناه:

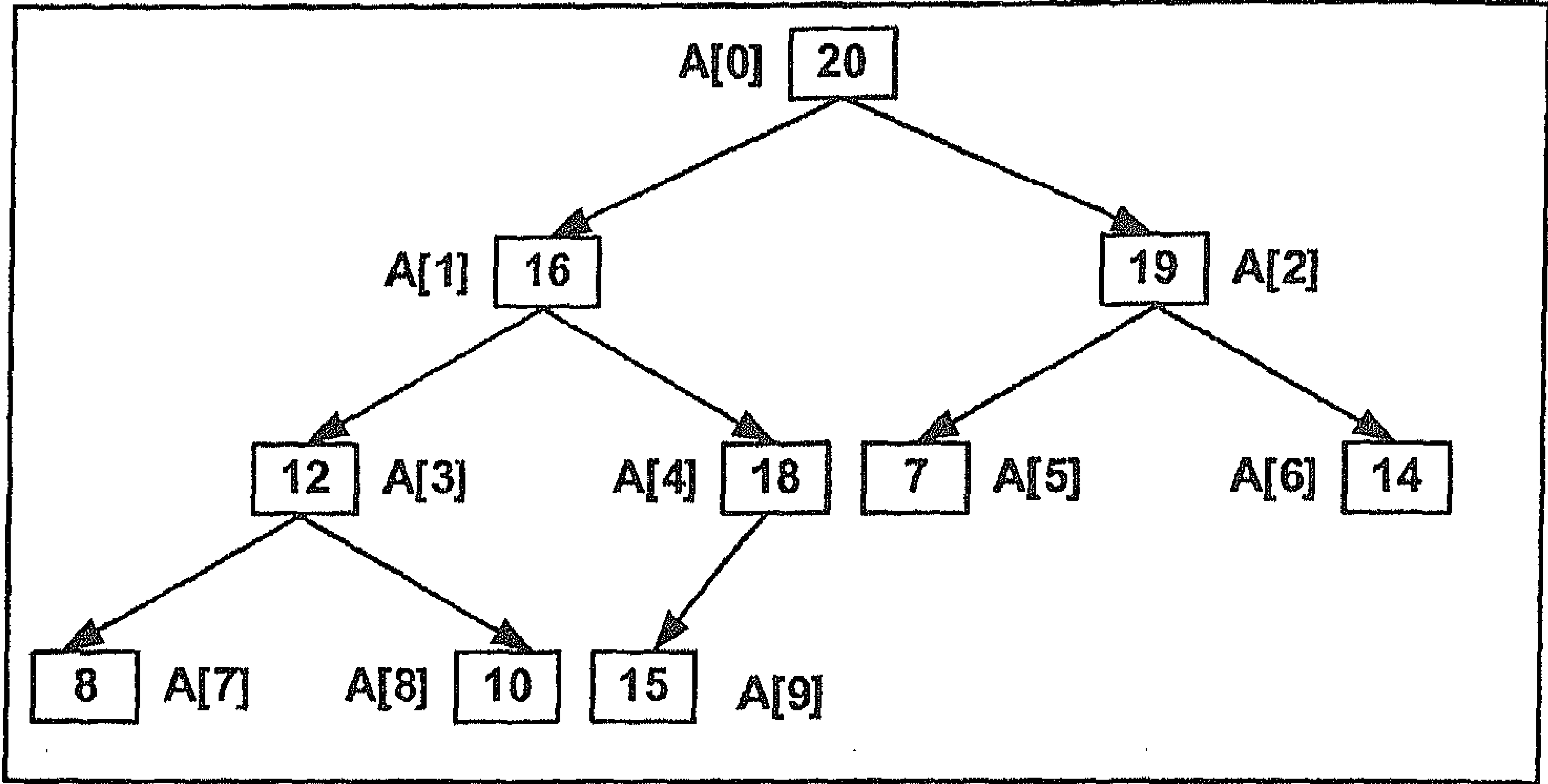


وفي الخطوة التالية نأخذ بعين الاعتبار العنصر الثامن (A[7] = 8). ونلاحظ أن قيمة هذا العنصر أصغر من العنصر الرابع (A[3] = 10) والذي يشكل الأهل للعنصر الجديد. ولذلك فإن العناصر الثمانية تشكل شجيرة ثنائية مقيدة ولا حاجة لإجراء أي تعديل على هذه العناصر.

بعد ذلك نفحص العنصر التاسع (A[8] = 12) ونجد أن قيمته أكبر من قيمة الأهل (A[3] = 10) ولذلك لا بد من إجراء تبديل بين محتويات هذين العنصرين لنحصل على الترتيب التالي:

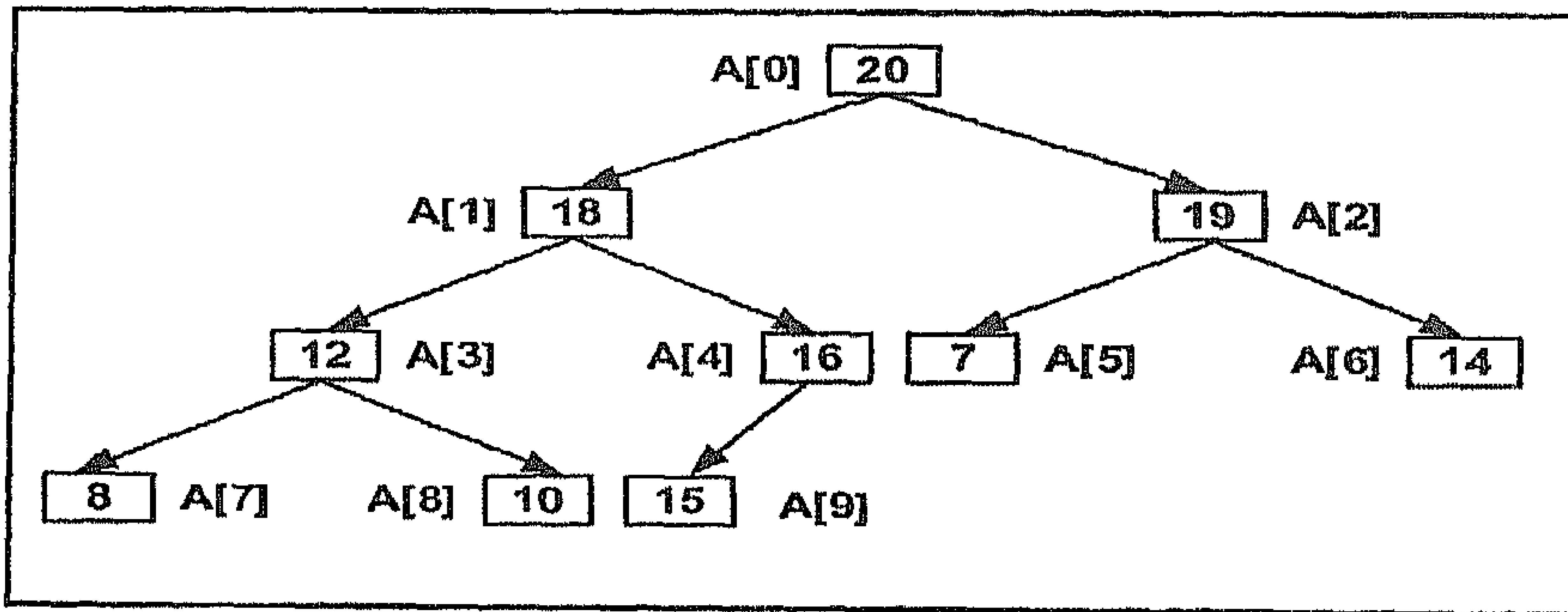


وأما بالنسبة للعنصر الأخير (A[9]=18)، فإن محتوياته أكبر من محتويات الأهل له (A[4]=15)، ولذلك لا بد من تبديل محتويات هذين العنصرين بحيث تصبح محتويات A[4] تساوي 18، ومحتويات العنصر A[9] تساوي 15 لنحصل على الترتيب التالي:



تلاحظ الآن أن قيمة العنصر A[4] (أي الابن الأيمن للعنصر A[1]) أكبر من قيمة العنصر A[1] (الذي يشكل الأهل للعنصر A[4])، ولذلك لا بد من إجراء التبديل على محتوياتهما، بحيث تصبح قيمة A[4] مساوية للقيمة 16 ومحتويات العنصر A[1] مساوية للقيمة 18.

وأخيراً، وعند مقارنة محتويات العنصر A[1] (الذي أخذ قيمةٍ جديدة هي قيمة A[4]) مع محتويات الأهل له أي A[0]، وكذلك مع محتويات الابن الأيسر له أي A[3]، تلاحظ أن العناصر العشرة أصبحت تشكل شجيرة ثنائية مقيدة كما هو مبين في الترتيب التالي:



وبعد انتهاء عملية بناء الشجيرة الثنائية المقيدة، تلاحظ أن أكبر قيمة موجودة في العنصر (أي جذر هذه الشجيرة). وهذا يشكل نهاية المرحلة الأولى (أي مرحلة بناء هيكل شجري ثنائي كامل مقيد) من الفرز المقيد.

ويمكن تحويل هذه الخطوات إلى دالة في إحدى لغات البرمجة مثل لغة سي++ على النحو التالي:

```
//temp is used as a non local integer variable
//Also A is used as a non local Array of integers
//N is the number of elements in the array
void BuildHeap()
  {int i,j,k;   //Indices for nodes
   int temp;
   for(k=0;k<N;k++)
    {i=k;         //Initialization i to index a node
     j=i/2;   //Initialize J as an index to parent node
     while (i!=0 && A[j]<A[i]) //The condition i!=0 makes sure that
                               //the process stops when reaching the
root
        {temp=A[j];             //switch contents of Node i
         A[j]=A[i];             //with that of its
         A[i]=temp;             //parent
         i=j;                   //set I to point to its parent
         if (i>1)
          j=i/2; }              //set J to point to its parent
    }
  }
```

يجب أن نذكر هنا أنه عند كل إضافة لعنصر جديد إلى شجيرة ثنائية مقيدة، يجب أن نفحص القائمة الجديدة بحيث تبقى شجيرة ثنائية كاملة مقيدة أيضاً.

وللتأكد من ذلك، لا بد من مقارنة محتويات هذا العنصر مع محتويات الأهل له (its parent). فإذا كانت قيمة الأهل مساوية أو أكبر من قيمة هذا العنصر فإن القائمة الجديدة تشكل شجيرة ثنائية كاملة مقيدة، وإلا فلا بد من تبديل قيم هذين العنصرين مع بعضهما البعض. وبعد هذا التبديل، يجب مقارنة المحتويات الجديدة للعنصر مع الأهل الجديد له، فإذا كانت محتويات الأهل الجديد ما زالت أقل يجب التبديل ثانية، وهكذا حتى تصل القيمة التي تم إضافتها إلى المكان المناسب في المصفوفة بحيث تشكل المصفوفة شجيرة ثنائية كاملة مقيدة.

أما الخطوة التالية في الفرز المقيد فتتمثل بتبديل العنصر الأكبر في الموقع الأول من المصفوفة والذي يمثل الجذر مع العنصر الموجود في الموقع الأخير من المصفوفة. وهذا يعني أنه عند انتهاء عملية الفرز، تترتب العناصر ترتيباً تصاعدياً. وبعد انتهاء هذه الخطوة نجد أن محتويات الموقع الآخر A[9] في المصفوفة تساوي 20 وهي أكبر قيمة في المصفوفة. أما محتويات العنصر الأول أي A[0] فتصبح مساوية للقيمة 15. وتصبح محتويات المصفوفة كما يلي:

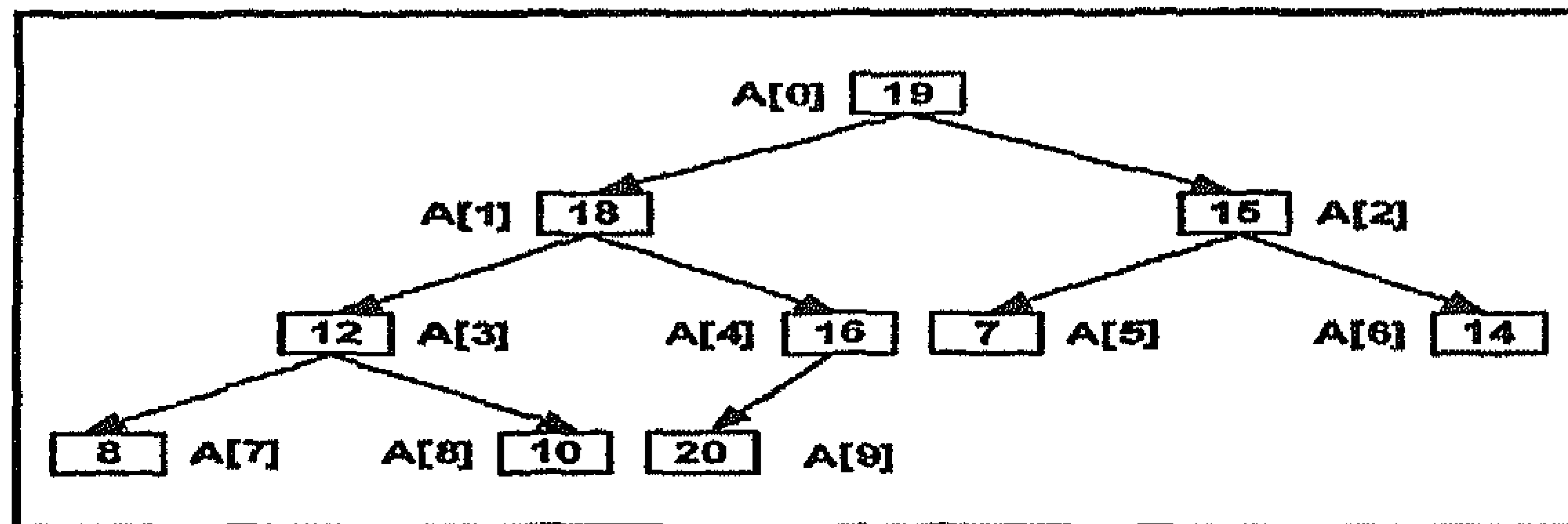
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
15	18	19	12	16	7	14	8	10	20

أما المرحلة التالية في الفرز المقيد فتتمثل في تعديل القيم المتبقية (أي A[0] إلى A[8]) بحيث تحصل على شجيرة ثنائية كاملة مقيدة مكونة من تسعة عناصر. ومن الممكن بعد عملية تبديل قيم العنصرين A[0] وA[9] أن تصبح العناصر A[0] إلى A[8] غير مرتبة على شكل شجيرة ثنائية كاملة مقيدة. ولذلك لا بد من نقل القيمة المخزنة في المكان الأول في المصفوفة (أي A[0]) إلى المكان المناسب، ولا بد من إحضار القيمة التالية في الكبر إلى الموقع الأول في المصفوفة.

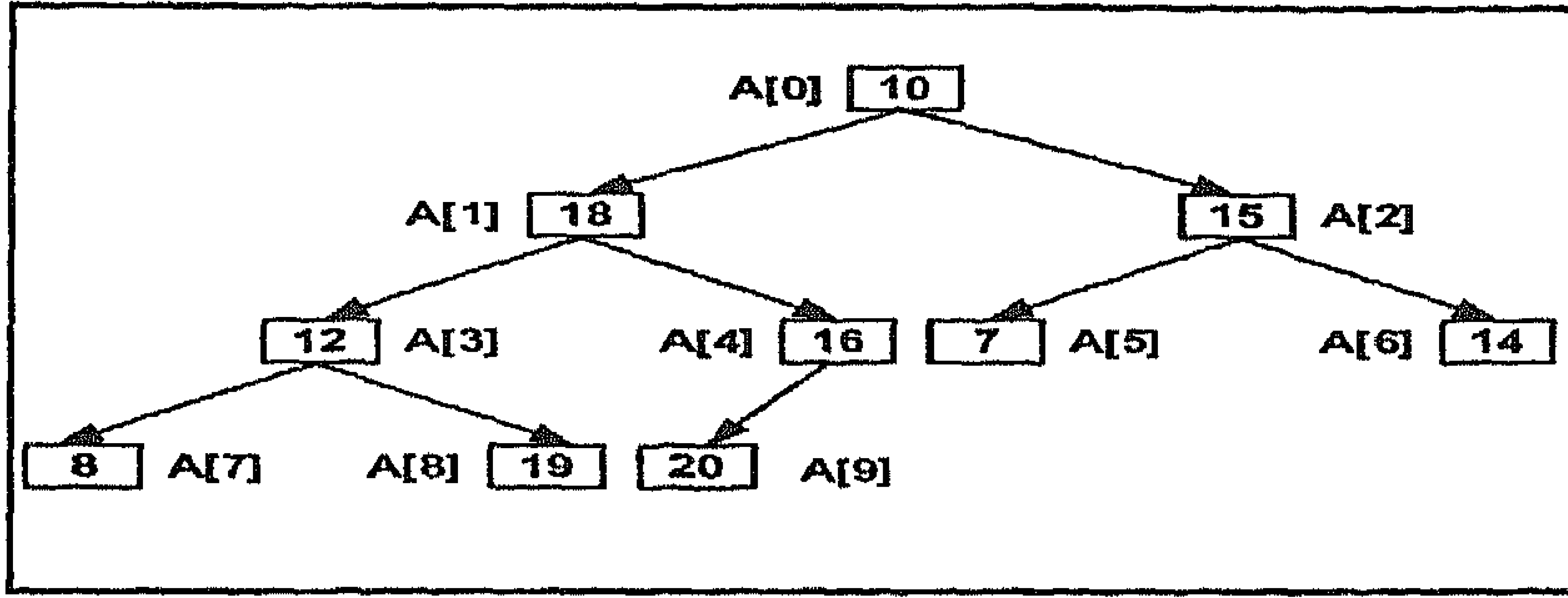
ولإنجاز ذلك، نقارن القيمة المخزنة في A[0] مع قيمة الابن الأيسر لها (أي قيمة A[1] وكذلك مع قيمة الابن الأيمن (أي قيمة A[2]).

فإذا كانت القيمة A[0] مساوية أو أكبر من قيم العناصر A[1] وA[2] فهذا يعني أن العناصر تشكل هيكلاً شجرياً ثنائياً كاملاً مقيداً، وإلا يجب مبادلة قيمة هذا العنصر مع قيمة العنصر الأكبر من الأبناء.

وبما أن قيمة A[2] (أي قيمة الابن الأيمن للجذر) والتي تساوي 18 أكبر من قيمة الإبن الأيسر للجذر وكذلك هي أكبر من القيمة المخزنة في الجذر، تجري عملية مبادلة القيم المخزنة في الموقع A[0] وA[2] وتحصل على الترتيب التالي للعناصر:



وبعد التبديل الأخير، تقارن قيمة العنصر A[1] مع قيم أبنائه، وبما أن قيمة A[2]أكبر من قيم أبنائه (أي A[5] وA[6]) فإن القيمة 15 هي في موقعها الصحيح وأن الشكل السابق يشكل شجيرة ثنائية كاملة مقيدة وأن الجذر يحتوي على أكبر قيمة من بين قيم العناصر A[0] إلى A[8] . وبهذا تأتي عملية التبديل بين A[0] وA[8] وتصبح قيم العنصرين A[8] وA[9] مرتبة. وتصبح محتويات المصفوفة كما هو مبين في الترتيب التالي:

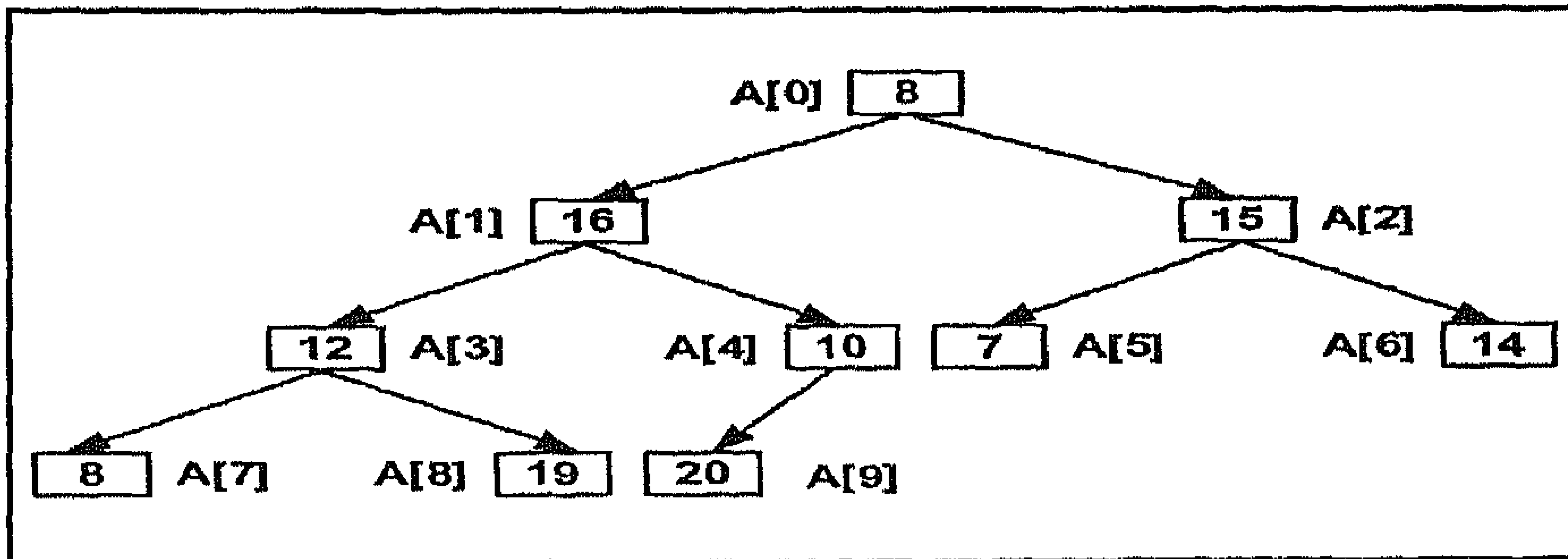


بعد ذلك يتم إعادة تنفيذ إجراء التعديل السابق وذلك لتعديل قيم العناصر من A[0] إلى A[7] لجعل هذه العناصر تشكل شجيرة ثنائية كاملة مقيدة، وفي هذا التعديل يتم إجراء التالي:

1. مقارنة A[0] مع القيمة الأكبر من A[1] وA[2] وبما أن A[1] أكبر من A[0] يتم تبديل A[0] وA[1] بحيث تصبح قيمة A[0] تساوي 18 وقيمة A[1] تساوي 10.

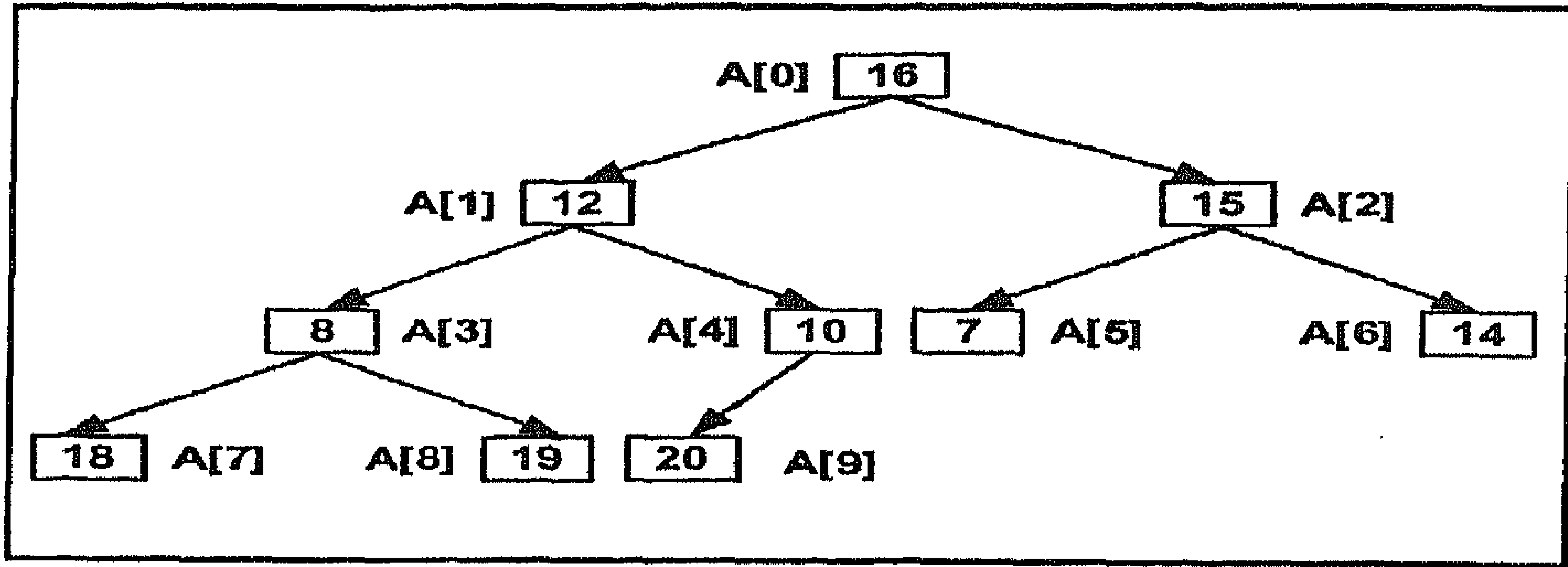
2. تقارن قيمة A[1] الجديدة مع القيمة الأكبر لأبناء هذا العنصر (أي مع القيمة الكبرى بين A[3] وA[4] وهي قيمة العنصر A[4] والتي تساوي 16، وبما أن 10 أصغر من 16، يجب تبديل محتويات هذين العنصرين بحيث تصبح قيمة العنصر A[1] تساوي القيمة 16 وقيمة العنصر A[4] تساوي القيمة 10.

وبما أن الابن الأيسر للعنصر A[4] هو الابن الوحيد له (وقيمته A[9]) وهو مرتب أصلاً، تتوقف هذه المرحلة وذلك باحتواء الجذر على القيمة الأكبر وهي 18. وهنا ينتهي إجراء التعديل على المصفوفة، ويتم الانتقال إلى عملية التبديل بين القيمة المخزنة في الجذر والقيمة المخزنة في A[7] بحيث تصبح العناصر الثلاثة الأخيرة في المصفوفة مرتبة ترتيباً تصاعدياً كما هو مبين في الترتيب التالي، والذي يعكس الوضع إلى هذه اللحظة:



بعد ذلك يتم تنفيذ عملية التعديل، للقيم المخزنة في العناصر A[0] إلى A[6] وتحويلها إلى شجيرة ثنائية كاملة مقيدة، وبما أن A[1] (القيمة الكبرى من أبناء A[0]) أكبر من قيمة A[0]، يتم تبديل هذين العنصرين بحيث تصبح قيمة A[0] مساوية للقيمة 16 وتصبح قيمة A[1] مساوية للقيمة 8. وأخيراً يتم تبديل قيم A[1] وA[3] وذلك لكون A[1] والتي تساوي 8 أصغر من A[3] والتي تساوي 12.

وعليه يمكن تمثيل المصفوفة على شكل شجيرة ثنائية كاملة حتى هذه اللحظة على النحو التالي:



وبما أن الابن الأيسر والابن الأيمن للعنصر A[3] قد احتل كل منهما المكان المناسب به في المصفوفة المرتبة ترتيباً تصاعدياً، تنتهي عملية التعديل. بعد ذلك تنتقل إلى عملية التبديل بين العنصرين A[0] وA[6].

وعند الوصول إلى هذه المرحلة (أي بعد تبديل العنصرين الأول والسابع) نلاحظ أن العناصر من A[6] إلى A[9] مرتبة وأن كل من هذه العناصر يحتل الموقع الخاص به عند ترتيب عناصر المصفوفة ترتيباً تصاعدياً.

أرجو، عزيزي الدارس، أن تكون عملية الفرز المقيد قد أصبحت واضحة في ذهنك، والتدريب التالي يطلب منك تكملة المثال (7) السابق وذلك ببيان الخطوات اللازمة للحصول على مصفوفة كاملة مرتبة.



تدريب (2)

أكمل المثال (7) السابق وذلك ببيان الخطوات اللازمة للحصول على مصفوفة عناصرها مرتبة ترتيباً تصاعدياً.

يمكن كتابة دالة التعديل على الهيكل الشجري لتحويله إلى هيكل شجري كامل مقيد (بحيث تبقى العناصر ممثلة في مصفوفة) بلغة سي++ كما يلي:

```
//This Function adjusts the nodes with indices from
//0 through m-2 so that the nodes again form a heapValue parameters
//m: Index of last node that was put into its final position
void AdjustHeap(int m)
 {int ElementToMove;  //Index of element to move
  int i;              //Index
  int s;                  //Child index
  bool Done;            //True when A[1] was not switched
  ElementToMove=A[1];
  Done=false;
  i=0;                 //Initialize i to root
  s=1;                 //Initialize S to left child of root
  while (s<m && !Done)
   {if((s+1)<m)          //f index for right child is less than m
   if (A[s+1]>A[s])
     s++;
   if (ElementToMove>=A[s])
     Done=true;
     else
      {//Switch contents of node i with that of its child
           A[i]=A[s];
      A[s]=ElementToMove;
      i=s;       //Set i to index to child
      s=2*i;    //Set s to index of left child of new node i
      }
   }
 }
```

وبعد كتابة دالة بناء الشجيرة الثنائية المقيدة الأولية التي تحتوي على جميع عناصر المصفوفة وإجراء تعديل الشجيرة بعد عملية التبديل بين الجذر والعنصر الأخير في المصفوفة الجزئية، لا بد من كتابة دالة الفرز المقيد الذي يستدعي الدالتين السابقين. والدالة التالية تبين ذلك:

```
//Declarations:
const N=100;  //Number of elements in the array to be Sorted

typedef int ArrayType[N];
/* HeapSort-Function to sort an array of integer Numbers in an
  ascending order using heapsort  parameter
 A: Entire integer array to be sorted
```

```
N: Number of elements in Data Array */
void HeapSort(ArrayType& A,int  N)
 {int temp;
 int j;
  //*********************************************
*******//
 //Insert BuildHeap and Adjust Heap procedures here.//
  //*********************************************
******//
 BuildHeap;                 //Create the initial N-element heap
 for (j=N-1; j>=0;j--)       //Exchange element J with root
  {temp=A[j];
  A[j]=A[1];
  A[1]=temp;
  AdjustHeap(j); //Adjust elements 1 through j-1
                //so that they reform a heap of j-1 elements
  }
 }
```

تحليل خوارزمية الفرز المقيد

إن كفاءة خوارزمية الفرز المقيد تعتمد على كفاءة الدالة BuildHeap، والتي تقوم ببناء الشجرة الثنائية المقيدة Heap، وعلى كفاءة الدالة HeapSort، والدالة AdjustHeap التي تقوم بتعديل الشجرة الثنائية بحيث يصبح أكبر عنصر فيها هو الأول (الجذر) أي تصبح شجرة ثنائية مقيدة (Heap).

لتحليل الدالة BuildHeap لاحظ، عزيزي الدراس، وجود دورانين في هذه الدالة الأول دوران For والثاني دوران While الداخلي، يتكرر الدوران for (n – 1) من المرات أما الدوران الداخلي While فيتكرر log2 n من المرات وذلك لأننا نقسم j على اثنين في كل مرة حتى يصبح 1، يتوقف الدوران وعليه فإن هذه الدالة هي O(n log2 n). لاحظ أن هذه الدالة تنفذ مرة واحدة إذ تستدعى مرة واحدة من قبل الدالة HeapSort. أما بالنسبة للدالة HeapSort فإنها تحتوي على دوران واحد يتكرر n - 1 من المرات ولكن في كل مرة تستدعى الدالة AdjustHeap، والتي تحتوي على دوران يتكرر log2 n من المرات، وذلك لأن الدوران الوحيد به (While) يبدأ بـ s: 2 وينتهي عندما تصبح قيمة s أكبر من m وأكبر قيمة لـ m هي n. لاحظ أيضاً أن العدد s يزداد بمقدار الضعف في كل مرة.

وعليه فإن الدالة HeapSort هي O(n log2 n)، وإذا أضفت تكلفة الشجرة الثنائية المقيدة فإن التكلفة الحقيقية هي 2n log2 n، وهذا يعني أن الخوارزمية تبقى O(n log2 n) عند تجاهل الثابت 2

2-3 طرق الفرز الخارجية (External Sorting Methods)

تعتبر طرق الفرز السابقة، التي تمت معالجتها من طرق الفرز الداخلية، حيث يفترض أن تكون البيانات مخزنة إما على شكل مصفوفة (Array) أو على شكل قائمة متصلة (Linked list)، أو على شكل شجيرة (Tree). وفي هذه الحالة يكون حجم البيانات ليس كبيراً جداً، ويمكن استيعابه في ذاكرة الحاسوب. ولكن عندما يكون حجم البيانات كبيراً ولا يمكن تخزينها في ذاكرة الحاسوب دفعة واحدة، لا بد من تخزينها في ملف مخزن على أحد وسائط التخزين الخارجية مثل القرص المغناطيسي أو الشريط المغناطيسي(Magnetic Disk). وفي هذه الحالة (أي عند إجراء عملية الفرز على البيانات المخزنة في ملف) يطلق على الفرز اسم الفرز الخارجي.

وهنا لا بد من الأخذ بعين الاعتبار خصائص الأجهزة المستخدمة للتخزين والوقت اللازم لنقل البيانات من الحاسوب وإليه على شكل قطاعات. ولإجراء عملية الفرز على قطاع من البيانات، لا بد من إحضار هذا القطاع إلى ذاكرة الحاسوب. بعد ذلك يتم نقل هذا القطاع إلى وحدة التخزين الخارجية بعد إجراء عملية الفرز ومن ثم إحضار قطاع آخر لإجراء عملية الفرز عليه. وهكذا يمكن فرز جميع القطاعات المكونة للملف المراد إجراء عملية الفرز عليه بإحضارها إلى ذاكرة الحاسوب قطاعاً قطاعاً، ومن ثم إجراء عملية الفرز. وبعد ذلك إعادة كل قطاع إلى وحدة التخزين الخارجية.

وأخيراً وبعد فرز جميع القطاعات، لا بد من دمجها معاً داخل الملف الذي يحويها.

وبما أن عملية الوصول إلى وحدات التخزين الخارجية لإحضار البيانات أو تخزينها يستغرق وقتاً أطول من معالجة هذه البيانات، فإن الفرز الخارجي يستغرق وقتاً أكبر مقارنة مع الفرز الداخلي.

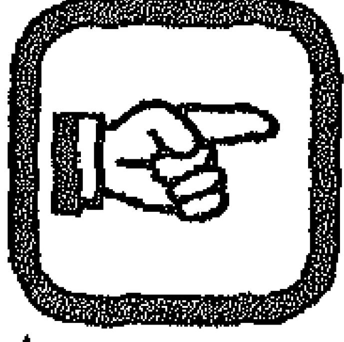
ويعتبر الفرز بالدمج (Merge Sort) من أهم طرق الفرز الخارجية وهو موضوع بحثنا التالي:

الفرز بالدمج (Merge Sort):

يستخدم هذا النوع من الفرز عند الحاجة لفرز كمية كبيرة من البيانات المخزنة في ملف على أحد وسائط التخزين الخارجية. ويتكون الملف عادة من عدد كبير من السجلات. ويمكن تلخيص خطوات هذا النوع من الفرز كما يلي:

• يتم أولاً تقسيم الملف الرئيس F إلى ملفين F1 وF2.

• بعد ذلك يتم مقارنة أحد الملفين بالآخر بحيث نقارن عنصراً من F1 مع عنصر آخر في F2. ومن ثم يتم دمجهما حسب الترتيب المطلوب، ويمكن توضيح ذلك والخطوات المتبقية في الفرز بالدمج بالمثال التالي:



مثال (8)

افرض أنك أعطيت ملفاً مخزناً على أحد وسائط التخزين الخارجي ويحتوي على البيانات التالية:

[4,8,5,1,6,33,25,10,13,21,23,40,16,60,30,47,32,11,37,14,15,20,7,90,70]

وترغب في ترتيبها تصاعدياً.

لإنجاز ذلك نلاحظ أن عدد العناصر الموجودة في الملف الأصلي هو 25 عنصراً. وعليه، نقسم هذا الملف في البداية إلى ملفين رئيسين F1 وF2 بحيث يحتوي الملف F1 على العناصر الإثني عشر الأولى، ويحتوي الملف F2 على العناصر المتبقية كما هو مبين أدناه:

- محتويات F1: [4, 8,5,1, 6,33, 25, 10, 13, 21, 23, 40]
- محتويات F2: [16, 60, 30, 47, 32, 11, 37, 14, 15, 20, 7, 90, 70]

في الخطوة التالية يتم أخذ عنصر من F1 مع عنصر آخر من F2 لتكوين زوج جديد من العناصر، بحيث يكون كل زوج من العناصر مرتباً. وبالأسلوب نفسه يتم تكوين جميع الأزواج المرتبة من العناصر، ويتم كتابة هذه الأزواج من العناصر بالتناوب إلى ملفين جديدين يمكن تسميتهما A1 وA2 . وفي نهاية هذه الجولة تكون محتويات A1 وA2 على النحو التالي:

- محتوياتA1: [(4,16),(5,30),(6,32),(25,37),(13,15),(7,23)]
- محتويات A2: [(8,60),(1,47),(11,33),(10,12),(20,21),(40,90),70]

وفي الخطوة التالية (أي في الجولة الثانية) يتم تكوين قطاعات طول كل منها أربعة عناصر، وذلك بأخذ زوج من A1 مع زوج من A2 وتكوين قطاع طوله 4 عناصر. بعد ذلك يتم ترتيب هذه العناصر حسب الترتيب المطلوب، ومن ثم تكتب هذه القطاعات إلى الملفات F1 وF2 كما يلي:

- محتويات F1: [(4,8,16,60),(6,11,32,33),(13,15,20,21)]
- محتويات F2: [(1,5,30,47),(10,14,25,37),(7,23,40,90),70]

وفي الخطوة التالية يتم تجميع كل ثمانية عناصر بحيث تؤخذ أربعة عناصر من الملف

F1 وتدمج مع أربعة عناصر من الملف F2 وتكتب العناصر الثمانية مرتبة إلى الملفين Al و A2 وبالتناوب. بعد نهاية هذه الجولة تصبح محتويات كل من الملفين A1 وA2 كما يلي:

- محتويات A1: [(1,4,5,8,16,30,47,60),(7,13,15,20,21,40,90)]
- محتويات A2: [(6,10,11,14,25,32,33,37),70]

وفي الجولة الرابعة يتم دمج القطاع الأول من الملف A1 (ويتكون من 8 عناصر) مع القطاع الأول من A2 (والذي يتكون أيضاً من 8 عناصر)، ويتم كتابة العناصر الناتجة والمرتبة بطريقة الدمج إلى الملف F1.

وكذلك يتم دمج القطاع الثاني من A1 (والذي يتكون من 8 عناصر) مع العنصر الوحيد في القطاع التالي من A2 للحصول على قطاع طوله 9 عناصر في الملف F2 بحيث تكون عناصره مرتبة حسب المطلوب.

وفي نهاية الجولة تصبح محتويات F1 وF2 كما هو مبين:

- محتويات F1: [1,4,5,6,8,10,11,14,16,25,30,32,33,37,47,60]
- محتويات F2: [7,13,15,20,21,23,40,70,90]

وفي الجولة الخامسة نجد أن F1 يحتوي على ستة عشر عنصراً، مكونة قطاعاً واحداً. والملف F2 يحتوي على تسعة عناصر تشكل قطاعاً واحداً أيضاً، بحيث أن محتويات كل قطاع منهما تتكون من عناصر مرتبة حسب المطلوب (أي ترتيباً تصاعدياً). وعند دمج هذين القطاعين تخزن جميع العناصر في الملف A1. وعند ذلك يصبح الملف A2 خالياً، وبما أن أحد هذين الملفين أصبح خالياً فإن عملية الفرز بالدمج تتوقف، ونحصل على محتويات A1 كما هو مبين:

[1,4,5,6,7,8,10,11,13,14,15,16,20,21,23,25,30,32,33,37,40,47,60,70,90]

في هذا المثال تم البدء بقطاع يتكون من عنصر واحد فقط، ومن الممكن زيادة كفاءة إجراء الفرز بالدمج إذا بدأت بقطاع يتكون من عدة عناصر حيث أنه من الممكن استيعاب مجموعة من العناصر في ذاكرة الحاسوب.

وعلى سبيل المثال، لو فرضنا أن الملف يحتوي على 10000 عنصر وأنه يمكن تخزين قطاع يتكون من 100 عنصر في الذاكرة في الوقت الواحد، فيمكن عندئذ حساب عدد الجولات اللازمة لفرز هذا الملف عن طريق المتباينة.

حيث أن n تمثل عدد الجولات فإذا كانت قيمة n تساوي 6 فإن قيمة التعبير (2^6*100) تساوي 6400 وهذا العدد أقل من عدد العناصر الإجمالي في الملف وهو 10000.

وأما بعد الجولة السابعة فيمكن ترتيب عدد من العناصر يساوي 2^7*100=12800.

وهذا العدد أكبر من العدد الفعلي للعناصر. ولذلك فإن ترتيب هذا الملف يحتاج إلى 7 جولات. ويمكن استخدام أي من عمليات الفرز الداخلية مثل الفرز السريع في كل جولة من الجولات.

ويجب الانتباه، عزيزي الدارس، إلى أن الفرز بالدمج يجب أن يعالج وسائط التخزين الخارجية، ولذلك فهو يعتمد على نوع وحدات التخزين المستخدمة. ولا يوجد متسع في هذا المكان لمعالجة خصائص وحدات التخزين الخارجية، حيث أنها تعالج في موضوعات أخرى.

ويلاحظ أن الفرز بالدمج الذي تم التعرض إليه في الشرح السابق قد بدأ بتقسيم الملف الرئيس إلى ملفين. وفي بعض الأجهزة يمكن تقسيم الملف إلى مجموعة من الملفات:

F1, F2, ..., Fr

يحتوي كل منهما على مجموعة من القطاعات، طول كل منها يساوي n من العناصر. ويمكن استخدام الفرز بالدمج على هذه الملفات حيث ينتج عن ذلك عدد مساو من الملفات الجديدة:

A1, A2, ..., Ar

يحتوي كل منها على قطاعات طول القطاع الواحد منها يساوي (n* r) من العناصر. وذلك لأن عملية إيجاد القطاع الجديد تتم عن طريق أخذ القطاع الأول من الملف الأول والقطاع الأول، والقطاع الأول من الملف الثاني، وهكذا إلى الملف الثالث. وبما أن طول القطاع الواحد يساوي n، فإن طول القطاع الواحد الجديد هو n*r، أي عدد الملفات مضروباً في طول القطاع.

ويجري الفرز بالدمج كما هو في المثال السابق، ويطلق على هذا النوع من الفرز، الفرز بالدمج متعدد الملفات.



تدريب (3)

افرض أنك أعطيت ملفاً يحتوي على العناصر التالية:

[22, 5, 70, 1, 60, 11, 55, 15, 40, 20]

وضح محتويات الملفات بعد تنفيذ كل جولة من جولات الفرز بالدمج لترتيب هذه العناصر ترتيباً تصاعدياً.

أسئلة التقويم الذاتي (1)

1. يمكن تقسيم الفرز إلى نوعين رئيسين، اذكرهما.
2. متى يستعمل كل نوع من النوعين السابقين ؟
3. أجب بنعم أو لا على العبارة القائلة بأن الفرز التجزئي هو فرز اختياري معدل.
4. بين القطاعات التي يمكن الحصول عليها في الفرز التجزئي عند تقسيم مصفوفة إلى 4 قطاعات علماً أن عدد عناصر المصفوفة هو 17 عنصراً.
5. ما قيمة K النهائية في الفرز التجزئي.
6. يعتبر الفرز التجزئي من أنسب أنواع الفرز من حيث معدل السرعة عند تطبيقه على القوائم الكبيرة العدد، أجب بنعم أم لا؟
7. متى تتوقف عملية زيادة المؤشر I وتنقيص قيمة المؤشر J في دالة الفرز السريع؟
8. حاول تطبيق دالة الفرز السريع بنوعيه (باستخدام الاستدعاء الذاتي وبدونه) على الحاسوب، ومن ثم أكمل البرنامج وطبقه على بعض الأمثلة. بعد ذلك حاول كتابة الدوال لوحدك بعد فهم عملها بالكامل.
9. لماذا سمي الفرز المقيد بهذا الاسم؟
10. ما ميزات الشجيرات الثنائية المقيدة؟
11. اكتب دالة بناء الشجيرة الثنائية المقيدة.
12. تأكد من فهمك لإجراء التعديل على الهيكل الشجري لتحويله إلى هيكل شجري كامل مقيد بكتابة هذا الدالة وتطبيقه على مثال.
13. اكتب الدالة اللازمة للقيام بعملية الفرز المقيد والتي تستدعي الدالتين السابقتين. في السؤالين 11 و12.
14. يعتبر الفرز بالدمج من طرق الفرز الداخلية. أجب بنعم أم لا.
15. يعتبر الفرز بالدمج من الأنواع الأكثر ملاءمة عندما تكون البيانات قليلة العدد ويمكن استيعابها في ذاكرة الحاسوب في وقت واحد. أجب بنعم أو لا.
16. افرض أنك أعطيت ملفاً يحتوي على القيم من 1 إلى 25. بين محتويات كل من الملفات(F1 وF2) ، و(A1 وA2) عند تنفيذ الخطوات الأولى من الفرز بالدمج.

4. طرق البحث Searching Methods

يعتبر البحث والفرز من أهم عمليات البرمجة على البيانات. وقد تطرقنا في البنود السابقة إلى عمليات الفرز، ونعالج الآن طرق البحث (أو الاستقصاء) الأكثر انتشاراً.

وتجدر الإشارة إلى أن البحث يتم عن عنصر معين ضمن مجموعة من العناصر. وفي حالة وجود هذا العنصر ضمن مجموعة العناصر تحدد عملية البحث موقع العنصر ضمن هذه العناصر.

وقد تكون البيانات مرتبة حسب مفتاح معين أو غير مرتبة. وفي أغلب الأحيان يتم البحث ضمن بيانات مرتبة وفق طرق الفرز الآنفة الذكر مما يساعد في إتمام عملية البحث بكفاءة أكثر. ومن الأمثلة على ذلك أنك تجد فرقاً شاسعاً في كفاءة عملية البحث في دليل الهاتف المرتب وبين عملية البحث عندما تكون الأسماء غير مرتبة. وكذلك الحال عند البحث عن كلمة في القاموس حيث الكلمات مرتبة فيه.

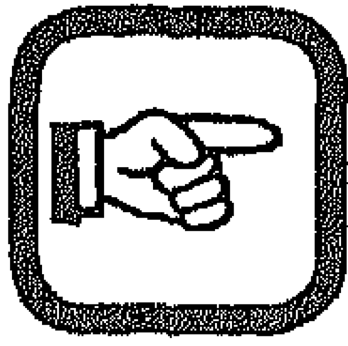
وفي هذا القسم نعالج، عزيزي الدارس، ثلاثة أنواع من طرق البحث هي البحث التتابعي، والبحث الثنائي، والبحث المفهرس.

1.4 البحث التتابعي (Sequential Search)

تعتبر هذه الطريقة من طرق البحث الأقل كفاءة والأكثر سهولة. وتستخدم للبحث في السجلات عندما تكون مخزنة دون أي اعتبار للترتيب، أو عندما تكون وسيلة التخزين لا توفر أي نوع من طرق الاسترجاع المباشر أو المفهرس (مثل الشريط المغناطيسي). وتتلخص هذه الطريقة بأخذ كل سجل من سجلات الملف على حدة، بدءاً بالسجل الأول في القائمة. وتتم مقارنة مفتاح البحث مع المفتاح الموجود في السجل الأول فإذا كشفت النتيجة عن وجود المفتاح في السجل يتوقف البحث ويعيد موقع هذا السجل، أما إذا لم يوافق مفتاح البحث مقابله في السجل فإن عملية المقارنة تنتقل للسجل الذي يليه، وهكذا حتى نهاية القائمة أو نجاح عملية المقارنة.

والخوارزمية التالية تبين البحث التتابعي:

```
//...Set Position to 0;
for (i=0;i<count;i++)
  if (List[1].Key=TestKey)
 //... Assign I to position;
```



مثال (9)

افرض أن لديك قائمة بالأسماء التالية:

AHMED, ALI, MOHAMMED, RAED, NAJIB, FAWAS, ABED

وأنك تريد استخدام طريقة البحث التتابعي للبحث عن اسم NAJIB إن كان موجوداً أم لا.

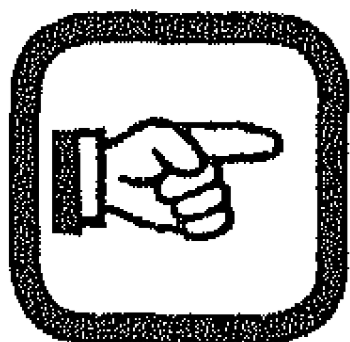
الحل:

حسب الخوارزمية في الصفحة السابقة فإن قيمة Position في البداية تساوي صفراً.

بعد ذلك تتم مقارنة قيمة مفتاح البحث TestKey (وهي هنا NAJIB) بأول عنصر من عناصر (List) فتجد أنه لا يساويه، فتنتقل إلى العنصر الثاني في List، كما هو مبين في الشكل (6). ثم تقارن العنصر الثاني بقيمة مفتاح البحث فتجد القيمتين غير متساويتين، وتتابع المقارنة مع باقي عناصر List حتى تصل إلى العنصر الخامس، فتلاحظ هنا بأن مفتاح البحث (Testkey) أصبح مساوياً لمحتويات العنصر الخامس من List، عندها تصبح قيمة (Position) هي (5)، وتنتهي عملية البحث. بإعادة القيمة (5) على أنها موقع السجل المطلوب.

1	AHMED	TestKey: NAJIB
2	ALI	
3	MOHAMMED	
4	RAED	
5	NAJIB	
6	FAWAS	
7	ABED	

الشكل (6)



مثال (10)

لو أجرينا تعديلاً على قيمة TestKey لكي تصبح (KHALED) فإنك تلاحظ أن عملية البحث المتتابع سوف تستمر بمقارنة TestKey مع جميع عناصر List حتى النهاية دون تغيير في قيمة (Position) (وهي صفر)، وهذا يعني أن العنصر غير موجود.

ويمكنك استبدال الخوارزمية السابقة بخوارزمية أكثر كفاءة، وذلك بتعديل شرط إنهاء البحث بحيث يشمل حالة إيجاد العنصر الذي يجري البحث عنه، بالإضافة إلى الوصول إلى نهاية القائمة في حالة عدم وجود العنصر والخوارزمية التالية توضح ذلك في القائمة.

```
Start with the First element in the list while more elements in List
 And Key not foundDo Get next element in List
If Key NOT found THEN
     Return record not found
ELSE
     Return record
```

والدالة التالية المكتوبة بلغة سي++ تبين دالة البحث التتابعي:

```
const N=10;
typedef struct Elements
  {int Key;
  Other...; //indicating other items in the record
  }elm;
typedef Elements Afile[N];

void Seqsearch(Afile A,int& i,int  N,int K)
 {/*This procedure search a set of records, may be in a file,
   with  key values A[1]. Key, A[2]. Key,..., A[n].
   Key, for a record such that A[i]. Key= K.
   If no such record is found, I is set to 0.*/
 A[0].Key=K;  //Dummy record to simplify the search
 i=N;            //which eliminates the need for End of file test
    while (A[i].Key!=K) i--;
 }// end of sequential search procedure
```

2.4 البحث الثنائي (Binary Search)

إن بساطة البحث التتابعي تعتبر من أهم خصائصه، وفي أسوأ الأحوال قد نحتاج إلى (N) من المقارنات للبحث في قائمة تحتوي على (N) من العناصر، وذلك لأن هذا النوع من البحث وكما أسلفت، يلائم القوائم المكونة من عناصر غير مرتبة، مع أنه يمكن أن يتعامل مع أي نوع من القوائم، سواءً كانت مرتبة، أو غير مرتبة، إلا أنه لا يستفاد من ترتيب العناصر في البحث التتابعي.

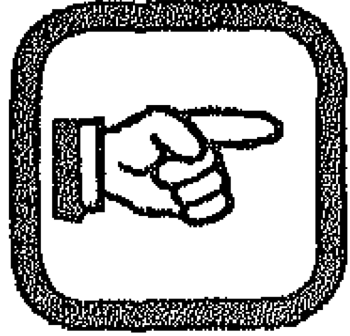
ولذلك إذا احتفظت، عزيزي الدارس، بعناصر القوائم مرتبة تصاعدياً أو تنازلياً أمكنك تحسين طريقة البحث بحيث أنه عند وجود (N) من العناصر تحتاج إلى من المقارنات

على الأكثر للوصول إلى الجواب. وهذا هو مبدأ البحث الثنائي الذي يعتمد على عناصر مرتبة. ويمكن تلخيص استراتيجية هذا البحث بالتالي:

- تبدأ عملية البحث في منتصف القائمة، فإذا كانت محتويات القائمة عند المنتصف أكبر من القيمة التي تبحث عنها فإنك في هذه الحالة تترك كل عناصر القائمة من المنتصف وحتى العنصر الأخير لأنه من غير الممكن أن يساوي أي منها العنصر الذي تبحث عنه،حيث أن جميعها أكبر منه. وهنا تقوم بالبحث في منتصف القائمة المحصورة بين أول عنصر والعنصر في منتصف القائمة الكلية. وتكرر هذه العملية حتى تجد العنصر الذي تبحث عنه، أو حتى ينتهي البحث دون إيجاد مثل هذا العنصر.

وتلاحظ هنا أن البحث الثنائي يقسم القائمة الرئيسة إلى قوائم فرعية بحيث أنه في كل مرة تحاول حصر العنصر بين طرفي القائمة الأصلية أو الجزئية. ولتوضيح مثل هذا النوع من البحث أسوق إليك المثال التالي:

مثال (11)



افرض أنك أعطيت القائمة التالية:

9	11	16	18	25	29	32	35
0	1	2	3	4	5	6	7
↑ First							↑ Last

والقيمة التي ترغب بالبحث عنها هي (25).

فكما هو واضح تحتوي هذه القائمة على ثمانية (8) عناصر مرتبة ترتيباً تصاعدياً.

في البداية يتم تحديد قيمة مؤشر البداية (First) وقيمة مؤشر النهاية (Last) للمصفوفة الكلية أو الجزئية التي قد يقع فيها العنصر الذي تبحث عنه. ولذلك تكون قيمة المؤشر First مساوية 1 وقيمة المؤشر Last مساوية 8 (أي عدد العناصر)، وذلك كما هو مبين:

First ↓			MidPoint ↓				Last ↓
9	11	16	18	25	29	32	35
0	1	2	3	4	5	6	7

بعد ذلك نقوم بتحديد نقطة الوسط حسب العلاقة التالية:

MidPoint = (First + Last) / 2
(8 + 1) = / 2
= 4

نلاحظ هنا أن قيمة العنصر عند الوسط (MidPoint) وهي 18 أصغر من القيمة التي تبحث عنها وهي 25. لذلك فإن العناصر من (1 حتى 4) لا تحتوي الرقم (25)، ومن المحتمل أن يكون هذا الرقم موجوداً في الجزء الآخر من القائمة، والذي يشمل العناصر الأربعة الأخيرة والتي تقع بين (MidPoint + 1) وLast. وبما أن الرقم الذي نبحث عنه أكبر من العنصر الموجود في الوسط فإنك بحاجة إلى تحريك مؤشر البداية إلى العنصر الذي يتبع الوسط، باستخدام العلاقة: First = MidPoint + 1 فتصبح القائمة كالتالي:

				First ↓			Last ↓
9	11	16	18	25	29	32	35
0	1	2	3	4	5	6	7
					↑ MidPoint		

ثم نعدل قيمة مؤشر المنتصف مرة أخرى كما يلي:

MidPoint = (5 + 8) / 2 = 6

وفي الخطوة التالية، نقارن العنصر الواقع في وسط القائمة الجديدة، (وهو العنصر السادس) مع القيمة التي نبحث عنها، فنلاحظ أن القيمة في الوسط أكبر من القيمة التي نبحث عنها (25<29). وفي هذه الحالة تتغير قيمة (Last) إلى العنصر الذي يأتي قبل الوسط مباشرة وذلك باستخدام العلاقة:

Last = MidPoint – 1

وهنا تلاحظ أن مؤشري البداية والنهاية متساويان في القيمة:

First = Last = 5

0	1	2	3	4	5	6	7
9	11	16	18	25	29	32	35
				↑ first=Last			

بعد ذلك نقوم بحساب الوسط الجديد كما يلي:

MidPoint = (5 + 5) / 2 = 5

وبذلك تصبح قيمة First, MidPoint, Last متساوية ومن جديد نقارن العنصر في الوسط (الموقع الخامس) مع الرقم (25) فنلاحظ أنه يساويه. فيكون الموقع الذي وجد

فيه العنصر مساوياً لقيمة MidPoint الأخيرة (وهي 5).

أما إذا كان العنصر غير موجود في القائمة، فإنه ينبغي إيقاف البحث عندما تصبح قيمة First أكبر من قيمة Last.

ولمعرفة مدى فعالية هذا النوع من البحث لاحظ أننا نحتاج إلى ثمانية مقارنات (كأسوأ حالة) لدى البحث في قائمة مكونة من ثمانية عناصر باستخدام البحث التتابعي. وفي المثال السابق فإنك تحتاج إلى (5) مقارنات. أما بالبحث الثنائي، فإنك تحتاج إلى ثلاث مقارنات فقط. وهذا الفرق قد لا يبدو كبيراً لأن عدد العناصر قليل، لكنه يتضح في حالة كون البحث ضمن عدد كبير جداً من العناصر. فإذا كان البحث في (1000) عنصر مثلاً، فإننا نحتاج في أسوأ الأحوال إلى (1000) مقارنة بالبحث التتابعي، بينما نحتاج إلى (log2 100) +1 (أي حوالي 11) مقارنة عند استخدام البحث الثنائي.

ويمكن كتابة دالة البحث الثنائي بلغة البرمجة سي++ على النحو التالي:

```
const StrSize = 20;
const N=10;
typedef char dataArray[StrSize];
typedef struct InfoStruct
  {int Key;
   dataArray data; //or any other appropriate type
  }myStruc;

int Actual; //Actual number of records to search
InfoStruct Info[N];

void BinSearch(int Key,dataArray& Item,bool& Found)
  {int First, Last, MidPoint; //Indices to elements
   First=1;
   Last=Actual;
   do
    {MidPoint=(First + Last)/2;
     if(Key>Info[MidPoint].Key)
       First=MidPoint +1;
       else
         if (Key<Info[MidPoint].Key)
                Last=MidPoint-1;
          else //Key found
            {Item=Info[MidPoint].data;
             Found=true;
            }
    }
   while (Found || Last<First);
   }
```

نشاط (1)



عند الوصول إلى العنصر i لإيجاد موقعه الترتيبي نلاحظ أن العناصر من A[0] إلى A[i-1]مرتبة. ولإيجاد الموقع المناسب للعنصر I ضمن العناصر A[0] إلى A[i-1] يمكنك استخدام البحث الثنائي بدلاً من البحث التتابعي حيث يعتبر البحث الثنائي أكثر كفاءة.

وبهذا التعديل نحصل على دالة جديدة. حاول كتابة هذه الدالة الجديدة باستخدام طريقة البحث الثنائي وذلك لفرز مصفوفة من العناصر العددية الصحيحة.

3.4 البحث المفهرس (Indexed Search)

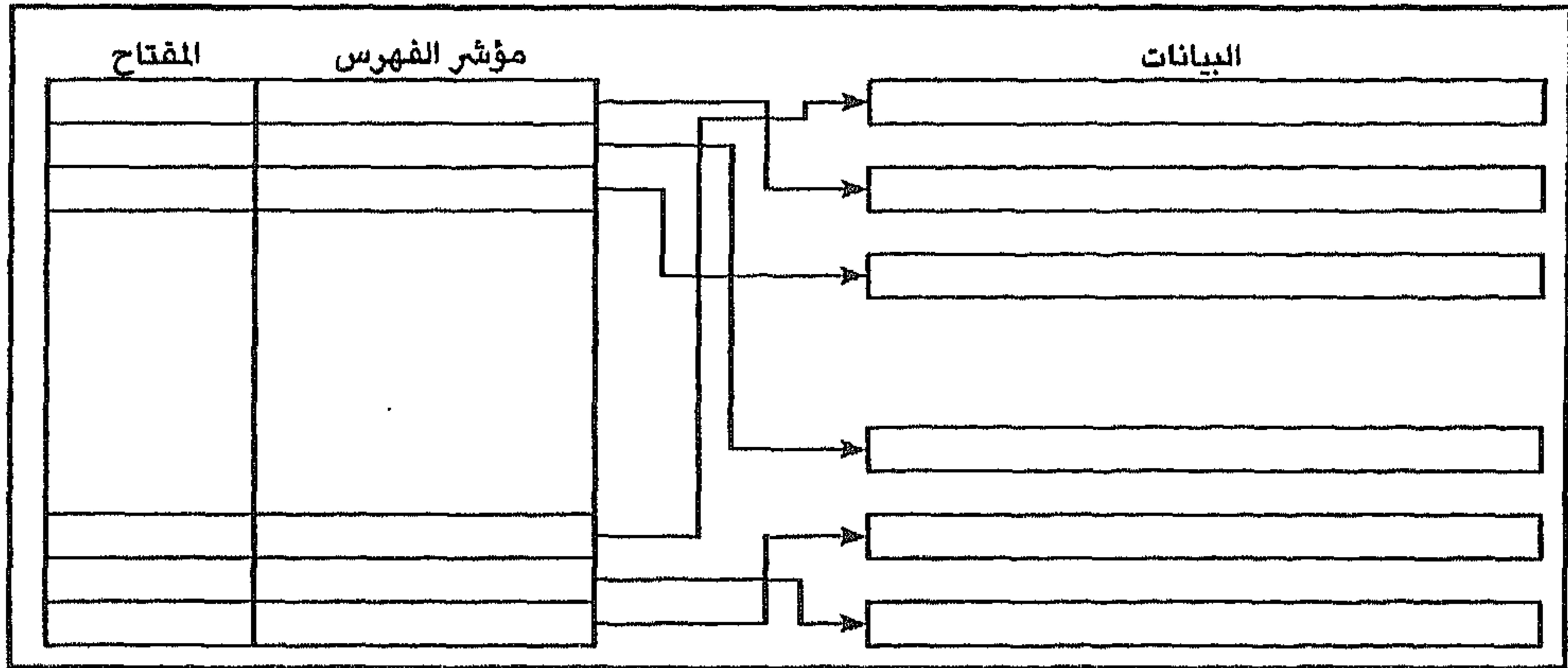
صمم هذا النوع من أنواع البحث للاستخدام مع الملفات التي تخزن على أجهزة التخزين المساعدة عشوائية الوصول، مثل القرص المغناطيسي (Magnetic Disk) . وسمي هذا النوع من أنواع البحث بهذا الاسم لتشابه طريقته مع الطريقة التي تتبعها للحصول على معلومات معينة (مثل العنوان) لشخص معين ضمن مجموعة من الأشخاص. وعلى سبيل المثال إذا رغبت في معرفة عنوان شخص ما، ليس من الممكن طرق باب كل شخص موجود لمعرفة ما إذا كان هو الشخص المطلوب أم لا.

والطريقة المستخدمة هي البحث في دليل الأشخاص عن الشخص المطلوب ومن ثم استخراج المعلومات المطلوبة (أي عنوانه في هذه الحالة) وذلك بمقارنة الاسم المطلوب مع قائمة الأسماء الموجودة في دليل الأشخاص. ويقال في هذه الحالة بأن اسم الشخص قد استخدم كمفتاح (Key). وفي بعض الأحيان يمكن التعامل مع أرقام خاصة بالأشخاص بدلاً من أسمائهم. وفي هذه الحالة يطلق على مثل هذا الرقم اسم المفتاح. وبالرجوع إلى المثال الحالي، وبعد مقارنة الاسم المطلوب مع قائمة أسماء الأشخاص، إما أن يتم إيجاد السجل الخاص بالشخص المطلوب أو ينتهي البحث دون إيجاد مثل هذا الشخص ضمن قائمة الأشخاص. وفي حالة نجاح عملية البحث نستطيع عندئذ الوصول إلى المعلومات المطلوبة عن الشخص وذلك بإتباع مؤشر خاص يشير إلى موقع المعلومات المطلوبة.

وفي حالة احتواء الملف المطلوب البحث فيه على معلومات ضخمة لكل سجل من سجلاته، تصبح عملية الفصل بين المفتاح والبيانات عملية مفيدة جداً. إن عملية الوصول إلى وحدات التخزين المساعدة، كما تعلم، أبطأ بكثير من عملية الوصول إلى الذاكرة الرئيسة في الحاسوب التي تتطلب أن توجد العناصر المطلوب مقارنتها فيها. ومن ثم فإنه

في حالة فصل المفاتيح عن محتويات السجل يمكن تجميع جميع المفاتيح (مع بعض المعلومات البسيطة مثل المؤشرات إلى بيانات كل سجل) على شكل فهرس حيث يمكن تجميعها في كتلة واحدة وتخزينها تخزيناً دائماً في ذاكرة الحاسوب الرئيسة. وفي هذه الحالة فإنه من غير الضروري إحضار البيانات الهائلة في السجلات والتي لا حاجة لنا بها لإنجاز عملية البحث لإيجاد العنصر المطلوب.

ويمكن تمثيل عملية الفصل بين المفاتيح (والمكونة على شكل فهرس) والبيانات كما هو مبين في الشكل (7) التالي:



الشكل (7)

تذكر، عزيزي الدارس، أن نجاح هذه الطريقة من طرق البحث يتطلب أن تكون قيم المفاتيح فريدة، ومن غير الجائز تكرار القيمة ذاتها لأكثر من مفتاح.

لذلك، يرتكز مبدأ البحث المفهرس على مقارنة قيم المفاتيح الموجودة في الفهرس مع مفتاح البحث للوصول إلى المفتاح المطلوب، وباتباع المؤشر الموجود بجوار المفتاح يمكن الوصول إلى البيانات الخاصة بالسجل، حيث يتم قراءة هذه البيانات قراءة واحدة.

ومن البين أن هذا النوع من طرق البحث يتطلب مساحة إضافية من الذاكرة تتسع إلى المؤشرات بالإضافة إلى المفاتيح.

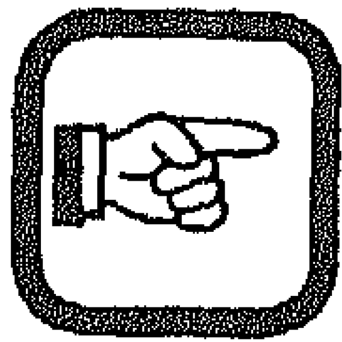
توجد عدة طرق للبحث المفهرس، تختلف عن بعضها البعض في كيفية تنظيم الفهارس نورد منه ما يلي:

1.3.4 البحث المفهرس التتابعي (Indexed Sequential Search)

غالباً ما يشار إلى هذا النوع من أنواع البحث باختصار ISAM وتمثل هذه الكلمة الأحرف الأولى من الكلمات الأربعة التالية: (Indexed Sequential Access Method).

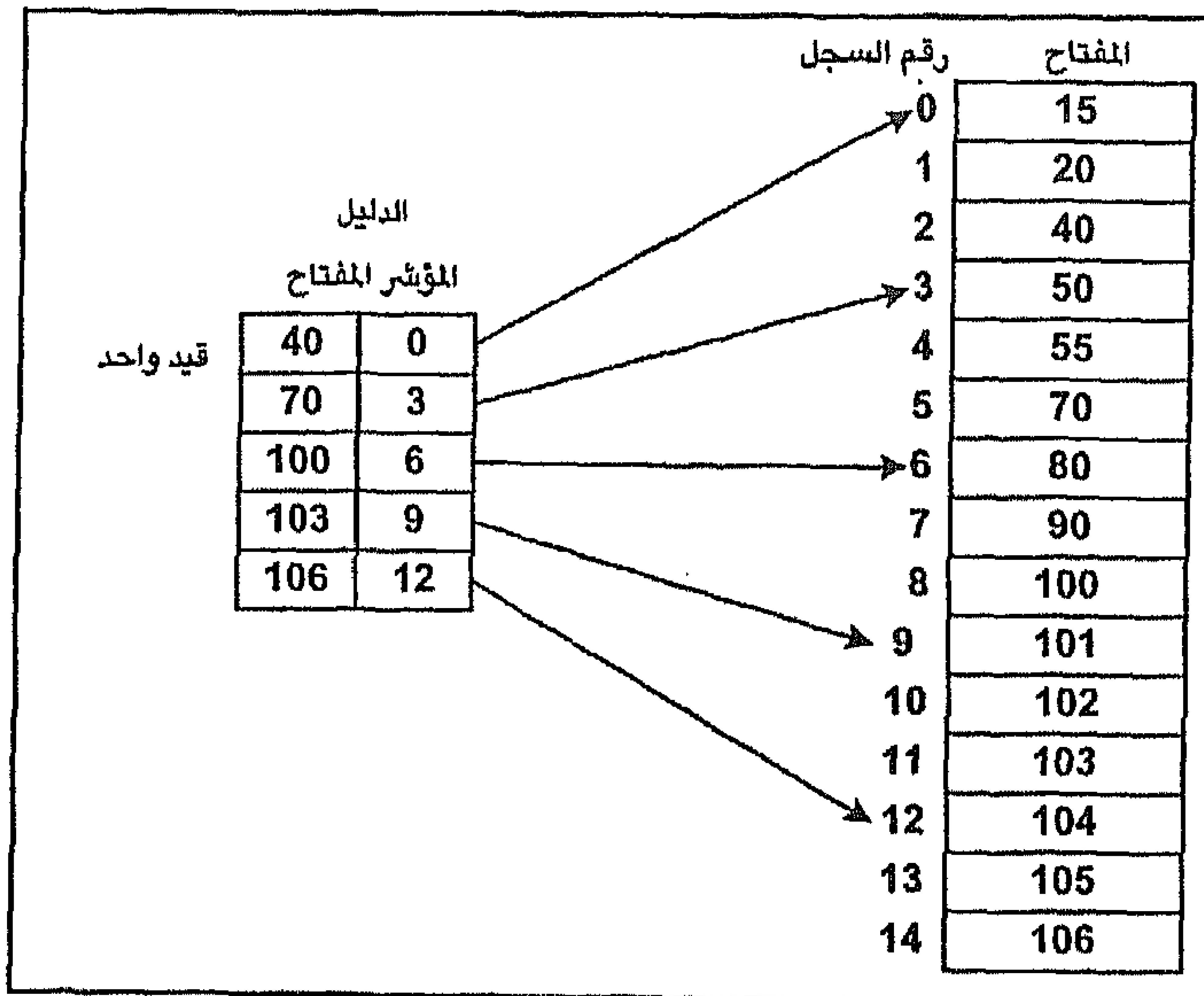
تتمثل هذه الطريقة في الأخذ بعين الاعتبار بعض الخصائص المهمة لوسائط التخزين المساعدة العشوائية (مثل القرص المغناطيسي) مثل معامل التكتل وحجم المسار وذلك للمساعدة في بناء فهرس جزئي. ومن الممكن لعمليات المقارنة على الفهرس أن لا تؤدي إلى الوصول إلى السجل المطلوب مباشرة، بل تؤدي إلى تحديد مجموعة من المفاتيح قد يكون مفتاح السجل المطلوب من بينها. بعد ذلك لا بد من إجراء البحث التتابعي على هذه المجموعة من المفاتيح لتحديد السجل المطلوب تمهيداً لقراءة بياناته.

وغالباً ما يطلق على الفهرس الجزئي اسم الدليل. ويمكن توضيح مفهوم عمل هذا النوع من طرق البحث بالمثال التالي:



مثال (12)

افرض أن المسار الواحد في قرص مغناطيسي يحتوي على ثلاثة سجلات وأن عدد السجلات الموجودة خمسة عشر سجلاً كما هو مبين في الشكل (8) التالي:



الشكل (8)

يحتوي الدليل في هذا المثال على خمسة قيود، يتكون القيد الواحد من جزئين هما المفتاح والمؤشر. ويشير كل مؤشر على بداية مجموعة من السجلات تحتوي كل مجموعة على ثلاثة منها.

يشير مؤشر القيد الأول في الدليل إلى العنصر الأول من المجموعة الأولى، ويشير مؤشر القيد الثاني إلى بداية المجموعة الثانية، وهكذا حتى القيد الأخير، حيث يشير مؤشر هذا القيد إلى بداية المجموعة الأخيرة من السجلات.

أما مفاتيح السجلات داخل المجموعة الواحدة منها فهي مرتبة بطريقة معينة (تصاعدياً في هذه الحالة)، وكذلك فإن جميع المفاتيح مرتبة أيضاً.

وأما بالنسبة لقيمة المفتاح الموجود في قيد الدليل فيكون الأكبر قيمة في المجموعة التي يمثلها هذا القيد.

لو أخذت، عزيزي الدارس، على سبيل المثال، المجموعة الأولى من السجلات، تلاحظ أن قيمة أكبر مفتاح فيها يساوي 40 وهذا يساوي أيضاً قيمة المفتاح في القيد الأول من الدليل.

وكذلك فإن قيمة المفتاح في آخر مجموعة من السجلات هو 106، وهذه القيمة مساوية لقيمة المفتاح في آخر قيد من قيود الدليل.

أما عن طريقة البحث، فافرض أنك ترغب في البحث عن السجل الذي يحتوي على مفتاح قيمته (102).

تبدأ عملية المقارنة بمقارنة هذا المفتاح مع المفاتيح الموجودة في الدليل وذلك لإيجاد مفتاح قيمته تساوي (أو أكبر من) هذه القيمة (أي قيمة مفتاح البحث وهي 102 في هذا المثال). وهنا تلاحظ أن هذا الرقم يساوي 103 وهو موجود في القيد الرابع مع قيود الدليل.

بعد ذلك يتم اتباع المؤشر (بأخذ قيمته) وتصل إلى السجل العاشر، والذي يبدأ بالمفتاح 101.

من هنا تبدأ المرحلة التالية، والتي تتمثل في عملية بحث تتابعي (تبدأ من السجل العاشر والذي يحتوي على المفتاح 101) لإيجاد السجل المطلوب أو الوصول إلى قيمة مفتاح أكبر من قيمة مفتاح البحث، وهذه إشارة إلى عدم وجود السجل ضمن السجلات المخزنة في الملف.

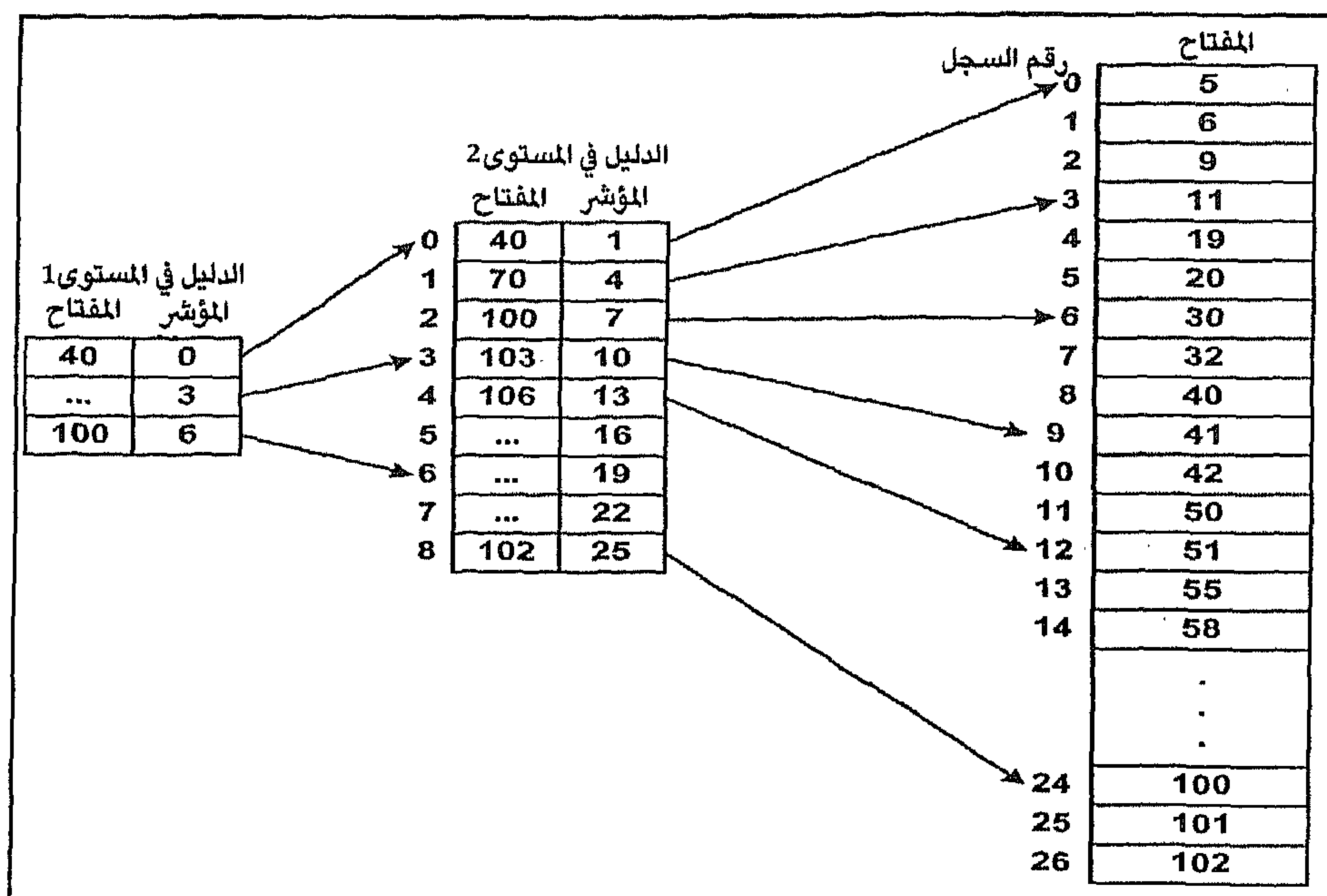
تبدأ عملية المقارنة بمقارنة 102 مع 101 (مفتاح العنصر الأول في المجموعة). وبما أن قيمة مفتاح البحث أكبر من قيمة مفتاح السجل الأول في هذه المجموعة تنتقل عملية

المقارنة إلى السجل الثاني في المجموعة حيث تلاحظ تطابق المفتاحين، مما يشير إلى أن هذا هو السجل المطلوب وتتوقف عندئذ عملية البحث.

لا شك، عزيزي الدارس، أنك لاحظت أن الدليل يحتوي على قيد واحد لمجموعة من السجلات ولا يوجد قيد لكل مفتاح. ولذلك لا بد من إجراء بحث تتابعي على مفاتيح كل مجموعة، ومن هنا جاءت التسمية بطريقة الوصول المفهرس التتابعي.

في هذا المثال كان عدد السجلات قليلاً نسبياً، وكذلك بالنسبة لعدد السجلات في المجموعة الواحدة. وكلما زاد عدد العناصر في المجموعة تجد أن معدل وقت البحث التتابعي قد زاد أيضاً، وهذا يعني أن عملية البحث قد تستغرق وقتاً أطول، مع أن مثل هذا الإجراء يقلل من عدد القيود الموجودة في الدليل.

وفي حالة احتواء الملف على بيانات كثيرة جداً فإنه قد يتم اللجوء إلى بناء مستويات من الفهارس الجزئية (أدلة) بحيث يشير الدليل الأول إلى دليل ثالث حيث يشير كل مؤشر في الدليل الأول إلى مجموعة من القيود في الدليل الثاني، وكل مؤشر في الدليل الثاني يشير إلى مجموعة من القيود في دليل آخر، وهكذا. وتشير المؤشرات لآخر دليل من هذه الأدلة إلى السجلات كما هو مبين في الشكل (9) التالي:



الشكل (9)

• وإذا تأملت الشكل السابق ستجد أن الدليل الأول يحتوي على مجموعة من المؤشرات يشير كل منها إلى العنصر الأول في مجموعة المؤشرات الموجودة في الدليل الثاني. وكل مؤشر في الدليل الثاني يشير إلى مجموعة من السجلات.

• لاحظ أن المؤشر (الأول) في الدليل الأول يشير إلى ثلاثة قيود من الدليل الثاني، وأن كل واحد من مؤشرات هذه القيود الثلاثة يشير إلى ثلاثة سجلات في الملف. فمثلاً يشير المؤشر الأول إلى السجلات (5 ,6 ,9) حيث أن العدد (9) هو أكبرها، ولذلك ورد في الموقع الأول من دليل المستوى الثاني وبجواره المؤشر الذي يشير إلى بداية هذه المجموعة من العناصر. لاحظ أيضاً أن الدليل في المستوى الأول يحتوي على مؤشرات إلى الدليل في المستوى الثاني، حيث يحتوي القيد فيه على القيمة الأكبر من كل ثلاث قيم في الدليل الثاني (40, 20, 9). ولذلك يكون المفتاح الوارد في الدليل الأول (وقيمته 40) أكبر القيم الثلاثة التي يشير إليها في دليل المستوى الثاني.

وتعتمد فعالية هذا النوع من أنواع البحث (Indexed Sequential) على العوامل التالية:

1. عدد مستويات الأدلة المستعملة.
2. مدة الاحتفاظ بهذه الأدلة في ذاكرة الحاسوب.
3. علاقة البيانات مع معاملات التكتل وحجم المسارات وحجم الاسطوانات.
4. وهنا تلاحظ أن هذا النوع من أنواع البحث غير فعّال للملفات ذات الأحجام الكبيرة، لأنه يقتضي أن تكون السجلات مرتبة تصاعدياً أو تنازلياً.

2.3.4 الفهرسة باستخدام الهياكل الشجرية الثنائية

(Binary Tree Indexing)

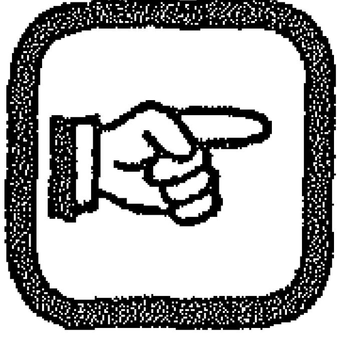
بالنسبة لعملية البحث على الهياكل الشجرية الثنائية فقد تم تغطيتها بشكل واسع في الوحدة السابقة "الهياكل الشجرية والمخططات". وسيتم التركيز هنا على الفهرسة باستخدام الهياكل الشجرية الثنائية.

إن المقصود بفهرسة الهيكل الشجري الثنائي احتواء كل موقع من مواقع الهيكل الشجري الثنائي على المفتاح الخاص بسجل ما، بالإضافة إلى مؤشر يشير إلى البيانات المتبقية من السجل.

ومن أهم الفوائد لهذا التنظيم ما يلي:

1. تتم عملية البحث بكفاءة عالية مساوية لكفاءة البحث الثنائية شريطة المحافظة على الهيكل الشجري الثنائي متزناً (height-balanced).
2. إعطاؤك القدرة على استعراض الهيكل الشجري الثنائي بحيث يمكن طباعة العناصر وفق ترتيب معين.
3. يمكن إجراء عمليات الإضافة والحذف بشكل مرن حركي (Dynamic)، مما يجعلها سهلة.

وتعتبر هذه الطريقة من أفضل الطرق المستخدمة في الفهرسة عندما تتسع ذاكرة الحاسوب الرئيسة للهيكل الشجري الثنائي بكامله. أما عندما يتعذر ذلك فإن الكفاءة تقل نظراً لتشتت توزيع عناصر الهيكل الشجري على أجزاء مختلفة من القرص المغناطيسي مما يؤدي إلى زيادة عمليات القراءة والكتابة من القرص المغناطيسي وإليه، والتي تستغرق وقتاً أطول.



مثال (13)

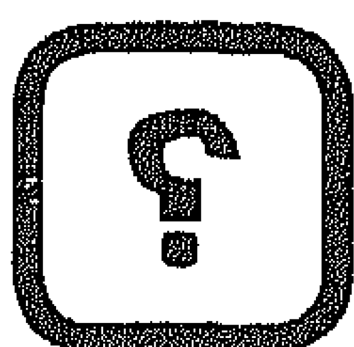
لو فرضنا على سبيل المثال أنك أعطيت قيم مفاتيح تم بناؤها على شكل هيكل شجري ثنائي كما يلي:

0	1	2	3	4	5	6	7	8	9
119	203	212	519	604	649	821	1300	1400	2600

وتود تحديد الموقع الذي يحتوي على المفتاح 1300. سوف يبدأ البحث بمقارنة القيمة 1300 مع القيمة في الموقع الخامس وذلك لأن عدد العناصر 10 وأن $\frac{9+1}{2} = 5$ وبذلك يبدأ البحث في الموقع الخامس. وبما أن محتوى الموقع الخامس هو 604 وأن 1300 أكبر من 604، يُقتصر البحث في الخطوة التالية على المواقع من 5 إلى 9. وتتم المقارنة مع الموقع الذي قيمته $\frac{9+5}{2} = 7$. وبما أن محتوى الموقع الثامن هو 1300 ينتهي البحث عن الفهرس في مقارنتين ويتم إيجاد الموقع الذي يحتوي على المؤشر الخاص ببيانات هذا الفهرس.

3.3.4 الفهرسة باستخدام الهياكل الشجرية نوع B - (B - Tree Indexing)

تستخدم هذه الطريقة من طرق الفهرسة للتغلب على بعض المشكلات الناجمة عن الطريقة السابقة، وذلك بتجميع عدد من عناصر الهيكل الشجري، والواقعة على مسار البحث نفسه في مجموعة واحدة، بحيث يتم تقسيم جميع العناصر إلى مجموعات. وللمزيد من المعلومات عن هذا النوع من الهياكل الشجرية يمكن الرجوع إلى الوحدة السابقة حيث تم توضيح ذلك بشيء من التفصيل.



أسئلة التقويم الذاتي (2)

1. اذكر ثلاثة أنواع رئيسة من طرق البحث.
2. ما أهم ميزات البحث التتابعي؟
3. اكتب دالة البحث التتابعي اللازمة للبحث عن قيمة ضمن مجموعة من القيم العددية الصحيحة مخزنة على شكل مصفوفة من النوع العددي وعدد عناصرها خمسة عشر.
4. ما ميزان البحث الثنائي؟
5. ما الشرط اللازم توفره في البيانات حتى يمكن تطبيق البحث الثنائي عليها؟
6. اكتب دالة البحث الثنائي وحاول تطبيقها على مجموعة من القيم الرمزية.
7. ما مبدأ البحث المفهرس؟
8. مع أي نوع من الأجهزة التي تخزن عليها الملفات يعتبر البحث المفهرس أكثر ملاءمة؟
9. اذكر ثلاث طرق للبحث المفهرس مع بيان كل نوع من هذه الأنواع.

5. الخلاصة

عزيزي الدارس، بعد أن قمت الآن بدراسة هذه الوحدة أوصيك بمراجعة الأهداف التعليمية المشار إليها في البند 2.1، وإذا شعرت أنك حققت تلك الأهداف تكون قد استوعبت موضوع هذه الوحدة، وإلا، حاول إعادة دراسة النص العلمي، بما في ذلك الأمثلة والتدريبات. وبعد ذلك يمكنك مقارنة حصيلة استيعابك لهذه الوحدة مع الملخص التالي:

تم تقسيم هذه الوحدة إلى ثلاثة أقسام أولهما مقدمة عامة عن الفرز والبحث، ثم قسم الفرز، ويليه قسم البحث.

وقد حاولنا في القسم الأول (ويعالج موضوع الفرز) بيان نوعين رئيسين من أنواع الفرز هما طرق الفرز الداخلية وطرق الفرز الخارجية.

وبالنسبة لطرق الفرز الداخلية، فقد تم معالجة ثلاثة أنواع رئيسة هي:

- الفرز التجزئي (Shell Sort)
- الفرز السريع (Quick Sort)
- الفرز المقيد (Heap Sort)

وبالنسبة لطرق الفرز الخارجية فقد تم معالجة أهم أنواع هذا الفرز وهو الفرز بالدمج (Merge Sort) والذي يستخدم عندما يكون عدد العناصر المطلوب فرزها كبير جداً ولا يمكن تخزينها بالكامل في ذاكرة الحاسوب في الوقت نفسه.

وأما بالنسبة لطرق البحث فقد عالجنا ثلاثة أنواع رئيسة هي:

- البحث التتابعي (Sequential Search)
- البحث الثنائي (Binary Search)
- البحث المفهرس (Indexed Search).

6. إجابات التدريبات

تدريب (1)

تتكون عناصر القائمة الثالثة من:

$A[2] = 38$
$A[2 + K] = A[2 + 7] = A[9] = 6$

ولا يمكن إضافة أي عنصر جديد إلى القائمة لكون (2K=16+2) وعدد العناصر في القائمة 16.

وبالطريقة نفسها يمكنك الحصول على عناصر المجموعة الرابعة وهي:

$A[3] = 19$
$A[3 + 7] = A[10] = 40$

وكذلك عناصر المجموعة الخامسة، وهي:

$A[4] = 36$
$A=[4 + 7]\ A[11] = 7$

وعناصر المجموعة الجزئية السادسة، وهي:

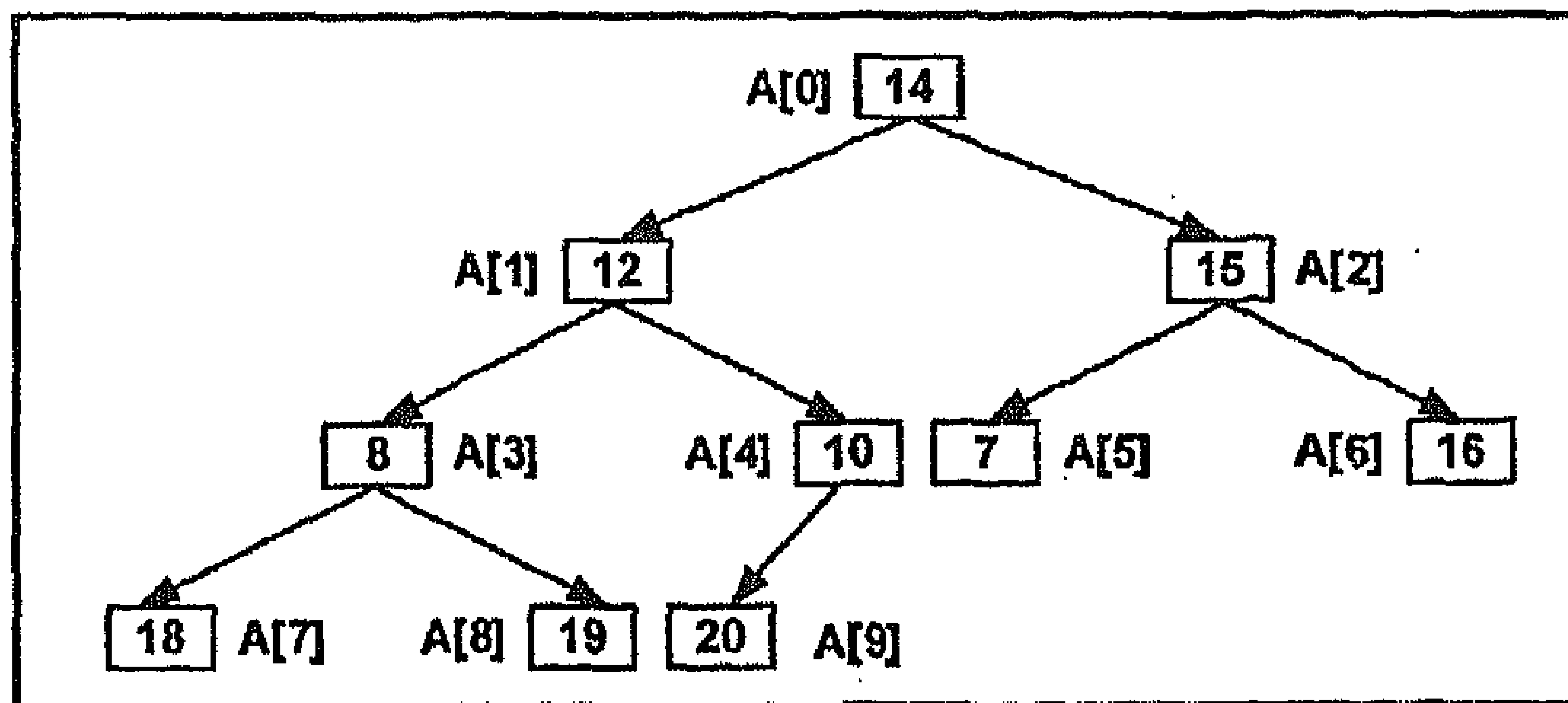
$A[5] = 45$
$A[5 + 7] = A\ [12] = 72$

وأخيراً عناصر المجموعة الجزئية الأخيرة، وهي:

$A[6] = 21$
$A[6 + 7] = A\ [13] = 90$

تدريب (2)

يمكن القول بأن محتويات العناصر في المصفوفة قد أصبحت على النحو التالي:



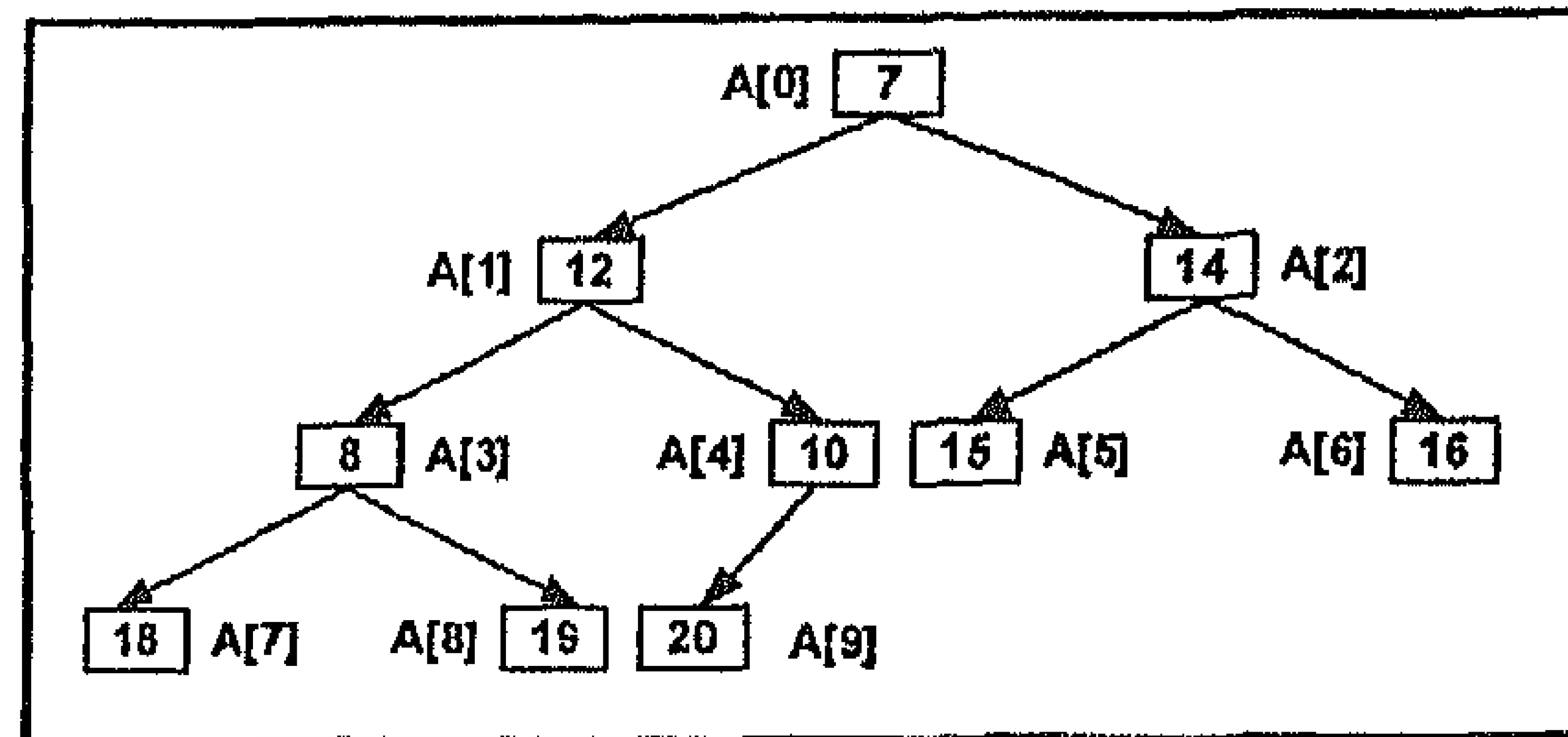
وفي الخطوة التالية يجب تعديل العناصر A[0] إلى A[5] بحيث تشكل هذه العناصر شجيرة ثنائية كاملة مقيدة. ويتم ذلك كما يلي:

1. بما أن قيمة الابن الأيمن (وهي أكبر من قيمة الابن الأيسر) للعنصر A[0] أكبر

من قيمة A[0]، يتم تبديل قيمة A[0] مع A[2]، بحيث تصبح قيمة A[0] تساوي 15، وتصبح قيمة A[2] تساوي 14.

2. يتم مقارنة A[2] مع A[5] وبما أن قيمة A[2] أكبر من قيمة A[5] تتوقف عملية التعديل.

بعد ذلك تبدل قيم العنصرين A[0] وA[5] بحيث تصبح عناصر المصفوفة من A[5] إلى A[9] مرتبة. والترتيب التالي للمصفوفة يوضح ذلك.

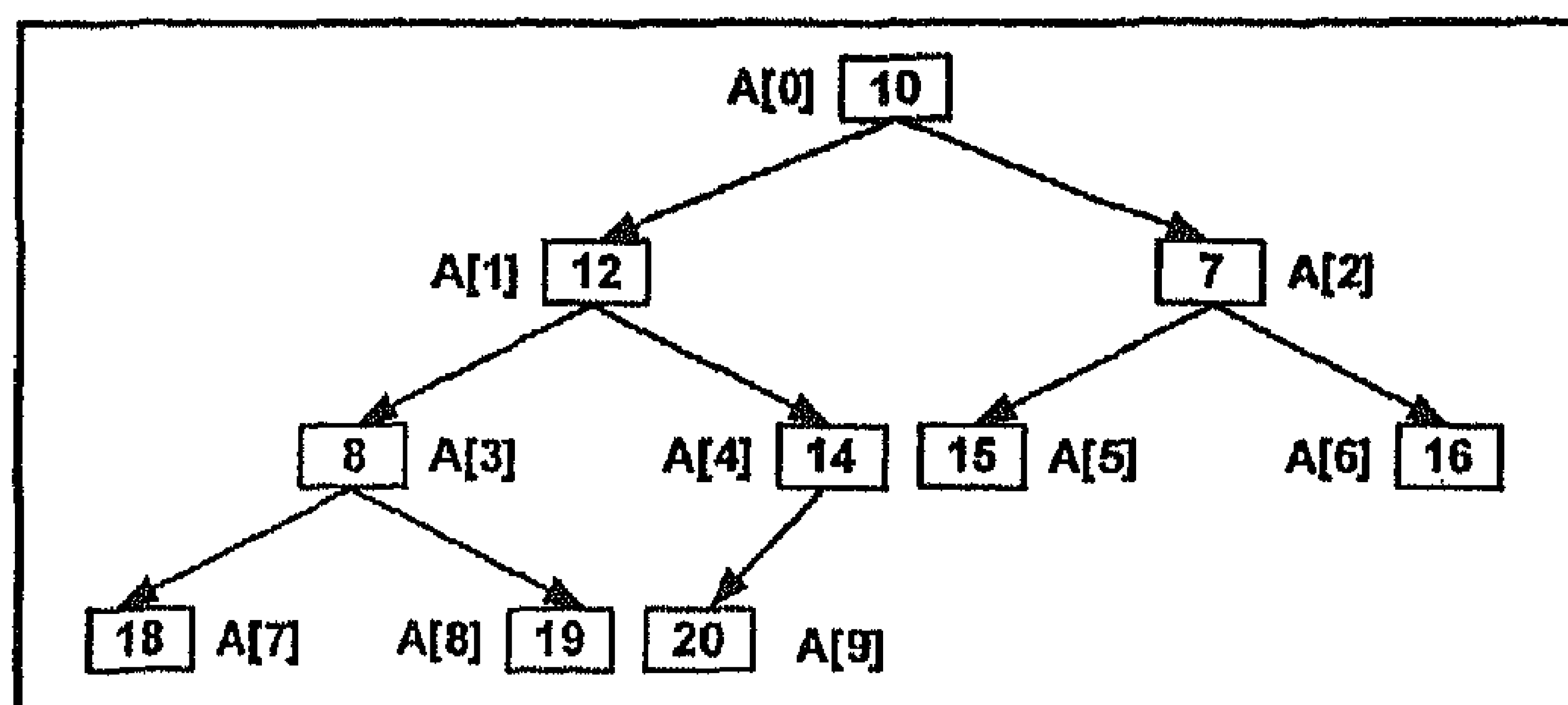


في الخطوة التالية يجب تعديل العناصر من A[0] إلى A[4] بحيث تنتج شجيرة ثنائية كاملة مقيدة.

وتبدأ هذه الخطوة بمقارنة A[0] مع القيمة الأكبر من A[1] وA[2] وبما أن A[2] أكبر من A[1] وA[0]، يتم تبديل محتويات العنصرين A[0] وA[2] (حيث أن A[2] هو الابن الأيمن للعنصر A[0]).

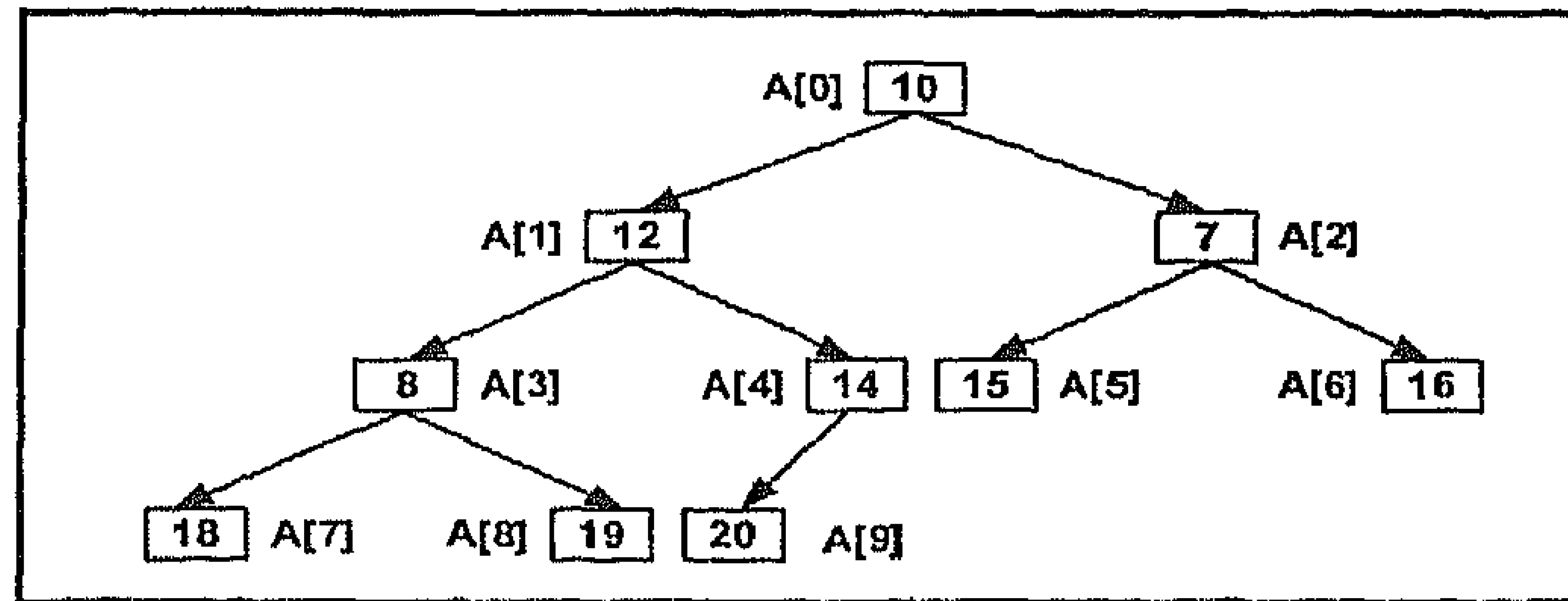
وبما أن أبناء العنصر A[2] قد رتبت، تتوقف عملية التعديل على العناصر A[0] إلى A[4] لتحويلها إلى شجيرة ثنائية كاملة مقيدة.

والخطوة التالية تشمل تبديل محتويات A[0] الجديدة بمحتويات A[4] وبذلك تصبح قيمة A[4] مساوية للقيمة 14 وقيمة A[0] مساوية للقيمة 10. ويمكن تمثيل المصفوفة حتى هذه اللحظة على النحو التالي:



بعد ذلك يجب تعديل العناصر A[0] إلى A[3] بحيث نحصل على شجيرة ثنائية كاملة مقيدة حيث تصبح قيمة A[1] أكبر من A[2] وكذلك أكبر من A[0]. ولذلك يتم تبديل قيم A[0] وA[1] بحيث تصبح قيمة A[0] مساوية للقيمة 12 وتصبح قيمة A[1] مساوية للقيمة 10. وبما أن A[1] أكبر من A[3]، تتوقف عملية التعديل وبذلك تشكل العناصر A[0] إلى A[3] شجيرة ثنائية كاملة مقيدة.

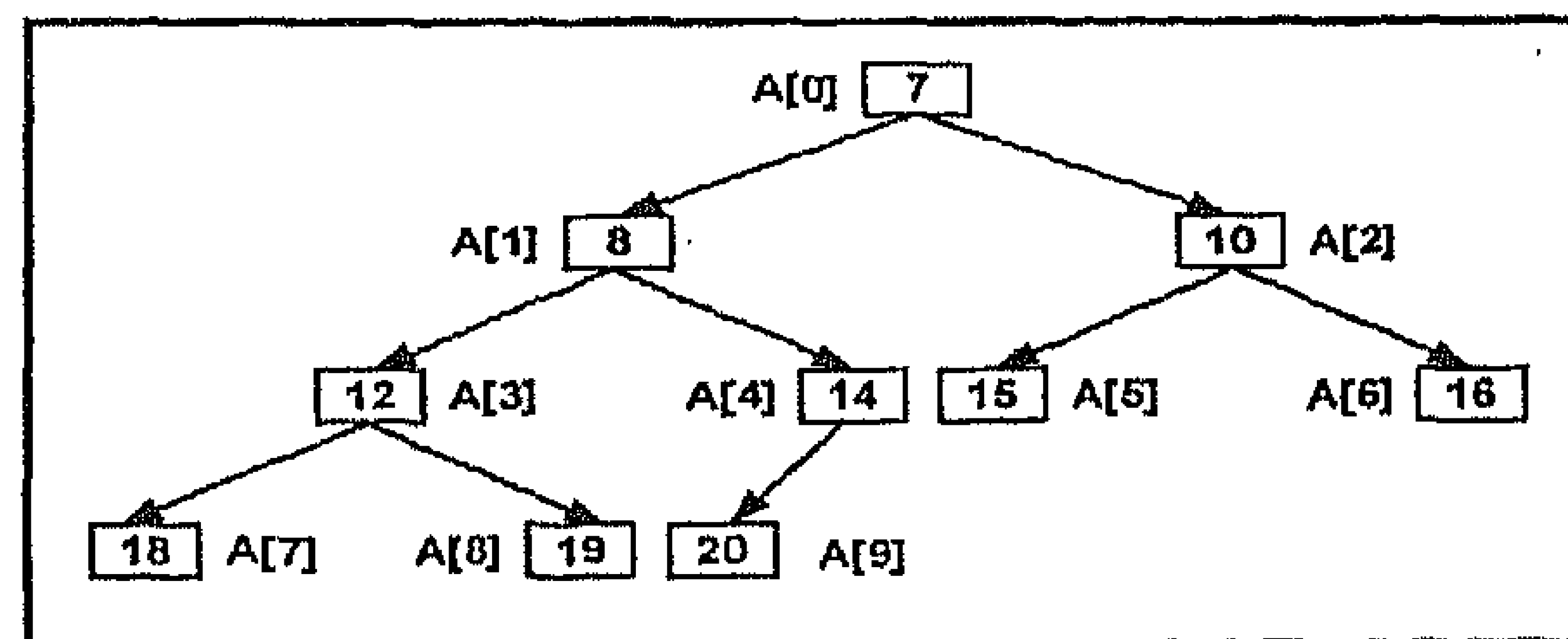
وفي الخطوة التالية تبدل محتويات العنصرين A[0] وA[3]، فتصبح محتويات عناصر المصفوفة كما هو مبين في الترتيب التالي:



وتصبح العناصر الثلاثة الأولى بحاجة إلى تعديل لكي تشكل شجيرة ثنائية كاملة مقيدة. ولإنجاز ذلك تبدأ بمقارنة A[0] مع A[1]، فتجد أن A[1] أكبر من A[2]، وبما أن A[1] أكبر من A[0]، يجري تبديل محتوياتهما بحيث تصبح محتويات A[0] مساوية للقيمة 10 وتصبح محتويات A[1] مساوية للقيمة 8.

وبهذا التبديل تنتهي عملية التعديل على الشجيرة بحيث تشكل العناصر الثلاثة من A[0] إلى A[2] شجيرة ثنائية كاملة مقيدة.

والخطوة التالية هي عملية تبديل بين A[0] وA[2] بحيث تصبح A[2] هي القيمة التالية في الترتيب. ويمكن تمثيل ذلك على النحو التالي:



وأخيراً يبقى العنصران A[0] وA[1] . وبما أن A[0] أصغر من A[1] فلا يتم أي تبديل بينهما، وتصبح القائمة مرتبة ترتيباً تصاعدياً.

تدريب (3)

في البداية يتم تقسيم الملف إلى ملفين رئيسين F1 وF2 حيث تكون محتويات كل منهما كما يلي:

- محتويات F1: [22, 5, 70, 1, 60]
- محتويات F2: [11, 55, 15, 40, 20]

● في الجولة الأولى يتم الدمج بأخذ عنصر من F1 مع عنصر من F2 (حيث يتم الاختبار من البداية) فنحصل على زوج جديد من العناصر، بحيث يكون كل زوج من العناصر مرتباً ترتيباً تصاعدياً. وتتكرر هذه العملية على جميع عناصر الملفين بحيث تكتب القطاعات الجديدة في الملفين A1 وA2 بالتناوب كما هو مبين:

- محتويات الملف:A1 [(11, 22), (15, 70), (20, 60)]
- محتويات الملف:A2 [(5, 55), (1, 40)]

● بعد إنهاء الجولة الثانية تكون محتويات الملفين F2, F1 كما يلي:

- محتويات F1: [(5, 11, 22, 55), (20, 60)]

● بعد إنهاء الجولة الثالثة تصبح محتويات الملفين A1 وA2 كما يلي:

- محتويات الملف:A1 [(1, 5, 11, 15, 22, 40, 55, 70)]
- محتويات الملف:A2 [(20, 60)]

● وبعد الجولة الرابعة تصبح محتويات الملفين F1 وF2 كما يلي:

- محتويات F1: [1, 5, 11, 15, 20, 22, 40, 55, 60, 70)]
- محتويات F2: خالية من العناصر.

وبما أن محتويات أحد الملفين أصبحت خالية، تنتهي عملية الفرز بالدمج، حيث يحتوي الملف F1 على جميع العناصر مرتبة ترتيباً تصاعدياً.

7. مسرد المصطلحات

- البحث (Searching): يطلق على عملية فحص مجموعة من العناصر لمعرفة ما إذا كانت قيمة ما موجودة ضمن هذه المجموعة أم لا. ويتم ذلك بمقارنة العناصر (أو المفتاح الموجود في السجل) مع مفتاح البحث (Search Key).

- البحث التتابعي (Sequential Search): يطلق هذا الاسم على طريقة البحث التي تبدأ بمقارنة مفتاح البحث مع أول عنصر في قائمة العناصر وتستمر عملية المقارنة هذه مع العناصر التالية حتى إيجاد عنصر أو سجل تكون فيه قيمة المفتاح مساوية لمفتاح البحث أو حتى يتم فحص جميع العناصر دون إيجاد العنصر المطلوب. ويستخدم البحث التتابعي مع البيانات غير المرتبة.

- البحث الثنائي (Binary Search): يستخدم مع العناصر المرتبة. ويبدأ البحث بمقارنة العنصر في وسط المجموعة مع مفتاح البحث (Search Key) . فإذا كان العنصران متساويان انتهت عملية البحث بإرجاع موقع هذا العنصر. أما إذا كانت قيمة مفتاح البحث أكبر من العنصر في الوسط فيقتصر البحث التالي في الجزء الواقع بين العنصر في الوسط وبين العنصر الأخير، وإلا فيقتصر البحث بين العنصر الأوسط والعنصر الأول. وتستمر هذه العملية حتى يتم إيجاد العنصر المطلوب وإعادة موقعه أو ينتهي البحث بدون أن يكون العنصر ضمن هذه العناصر.

- البحث المفهرس (Indexed Search): يستخدم هذا النوع من طرق البحث للبحث عن قيمة معينة ضمن قائمة من المفاتيح مرتبة على شكل فهرس (Index).

- الفرز (Sorting): يطلق على عملية ترتيب مجموعة من العناصر (عددية أو رمزية أو سجلية وما إلى ذلك) وذلك وفق ترتيب معين (تنازلي أو تصاعدي). وفي حالة كون العناصر من النوع السجلي يتم الفرز حسب حقل أو أكثر من حقول السجل يطلق عليها اسم المفتاح (Key).

- الفرز التجزئي (Shell Sort): في هذا النوع من طرق الفرز تقسم مجموعة العناصر المراد إجراء الفرز عليها إلى قطاعات يتم فرز كل منها على حدة وذلك باستخدام الفرز الإدخالي.

- **الفرز الخارجي (External Sort)**: يطلق هذا الاسم على طرق الفرز التي تستخدم مع البيانات المخزنة في ملفات على وسائط التخزين المساعدة مثل الأشرطة والأقراص المغناطيسية. وتتميز طرق الفرز الخارجية باستخدامها على عدد كبير من البيانات بحيث لا يمكن تخزينها بالكامل في ذاكرة الحاسوب في وقت واحد.

- **الفرز الداخلي (Internal Sort)**: يطلق على طرق الفرز التي تعمل على عدد قليل من العناصر بحيث تبقى هذه العناصر داخل ذاكرة الحاسوب طيلة فترة تطبيق الفرز عليها.

- **الفرز السريع (Quick Sort)**: في هذا النوع من طرق الفرز الداخلي يتم اختيار عنصر من العناصر كمحور (Pivot) لتقسيم العناصر (أو مفاتيح السجلات). وغالباً ما يتم اختيار العنصر الوسيط لهذا الغرض. وفي المرحلة التالية لعملية تحديد العنصر، يتم تحريك العناصر إلى يمين هذا العنصر ويساره، بحيث تكون جميع العناصر المساوية أو الأصغر منه في القيمة إلى يسار هذا العنصر، وأما العناصر المتبقية فتكون إلى يمينه. وتتكرر عملية اختيار عنصر آخر في المجموعة الأولى، ويتم فصل العناصر كما حدث في السابق، وتتكرر هذه العمليات حتى يتم في النهاية ترتيب جميع العناصر.

- **الفرز المقيد (Heap Sort)**: في هذا النوع من أنواع الفرز الداخلي تتم عملية الفرز على مرحلتين، في المرحلة الأولى نقوم ببناء هيكل شجري ثنائي كامل مقيد (Heap) بحيث تكون القيمة المخزنة في أي موقع من مواقع الهيكل الشجري أكبر من أو مساوية للقيم المخزنة في مواقع الأبناء إن كان هناك أبناء لهذا الموقع. وفي المرحلة الثانية يحدد العنصر التالي في الترتيب بحيث يصبح مخزناً في جذر الشجيرة، ومن ثم يعدل الهيكل الشجري الثنائي ويحول إلى هيكل شجري مقيد. وتتكرر المرحلة الثانية إلى أن تترتب جميع العناصر.

- **مفتاح البحث (Search Key)**: قيمة تستخدم لمقارنتها مع العناصر أو قيم المفتاح في سجلات العناصر المطلوب البحث فيها عن سجل معين لتحديده ضمن مجموعة من السجلات وذلك للوصول إلى محتويات السجل.

8. المراجع

1. Kruse, Robert L., Data Structures and Program Design. Englewood Cliffs (USA): Prentice-Hall, 1984.
2. Lipschutz, Seymour, Theory and Problems of Data Structures. New York: McGraw-Hill, 1986.
3. Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison- Wesley, 1999.
4. Lewis, T. G.; Smith, M.Z., Applying Data Structures. Atlanta (USA): Houghton Mifflin, 1976.
5. Tenenbaum, Aarom M.; Augenstein, Moshe J., Data Structures Using Pascal. Englewood Cliffs (USA): Prentice-Hall, 1981.

هذا الكتاب

إن موضوع تركيب البيانات على درجة بالغة الأهمية في علم الحاسوب ، حيث أنه يناقش الطرق المختلفة لتنظيم البيانات داخل الحاسوب . ومن المهم جداً ، عزيزي القارئ ، اختيار الطريقة المناسبة لتنظيم البيانات لعدة أسباب منها:

- أن اختيار الطريقة المناسبة لتمثيل البيانات يؤدي إلى استخدام خوارزميات حل أكثر كفاءة (من حيث وقت التنفيذ وحجم ذاكرة الحاسوب المستخدمة. ومن هنا يكون الترابط بين موضوع تراكيب البيانات وتحليل الخوارزميات ، (لمعرفة وقت التنفيذ وحجم ذاكرة الحاسوب اللازمين) وكثيراً ما يقال إن :

البرنامج = تركيبة بيانات + خوارزمية الحل.

-أن اختيار الطريقة المناسبة لتمثيل البيانات يجعل البرنامج أيسر للكتابة، حيث تكون خوارزمية الحل أوضح وأقرب إلى الواقع مما يؤدي إلى تقليل الوقت والجهد اللازمين لكتابة البرنامج.

-أن وضوح ومنطقية طريقة الحل تؤدي إلى برامج واضحة يسهل فهمها وتعديلها عند الحاجة في وقت قصير.



الشركة العربية المتحدة للتسويق وا

P.O Box: 203 Heliopolis 11757 Cairo - Egypt
Mobile: 002-010-1763677 Mobile: 002 - 010 - 3401184
E-mail: info@uarab.net u_arab@yahoo.com Web : www.uarab.net